2
Fundamentação Teórica

2.1
Serviços

A Arquitetura Orientada a Serviços (SOA) é uma evolução da computação distribuída, na qual as lógicas de negócio são encapsuladas na forma de serviços de maneira modular, de forma que possam ser invocados por clientes.

A seguir são apresentadas algumas características da Arquitetura Orientada a Serviços (Papazoglou and Heuvel 2007):

1. **Neutralidade de implementação**: Serviços implementados em diferentes tecnologias podem se comunicar, visto que em SOA, o que importa é a interface do serviço;

2. **Fraco acoplamento**: Em SOA cada serviço, deve ser entendido como um elemento independente, da mesma forma que um componente é independente na Arquitetura Orientada a Componentes (Szyperski 1997);

3. **Granularidade**: Os serviços em SOA devem ser tomados como elementos de baixa granularidade, pois deve-se ter uma visão tão abstrata quanto possível, ocultando detalhes referentes às ações e interações realizadas internamente pelo serviço;

4. **Configuração flexível**: Os sistemas devem oferecer suporte para configurações de maneira flexível e dinâmica.

Devido as características supracitadas, a Arquitetura Orientada a Serviços tem como benefícios um alto nível de abstração na construção de sistemas de grande porte, desenvolvimento de ferramentas capazes de gerenciar todo o ciclo de vida da aplicação, entre outros.

Pelo fato de SOA ser uma arquitetura, não existe um implementação padrão. Ela pode ser implementada utilizando diversas tecnologias, como Corba(Aberdeen and et. al. 1997), Web Services (Papazoglou 2007), etc. Sendo que neste trabalho nós utilizamos a tecnologia de *Web Services*.

2.1.1 *Web Services*

Um *Web Service* (WS) é um componente de software que pode ser acessado e usado remotamente. A comunicação dos *Web services* é construída sobre tecnologias padrões da Internet, como XML(W3C 2003b), HTTP(W3C 2003a), e outros protocolos que dão suporte a interoperabilidade. Usando protocolos padronizados, *Web services* permitem que desenvolvedores criem aplicações que são compatíveis com diferentes linguagens de programação, sistemas operacionais, plataformas de hardware e que são acessíveis de qualquer localização geográfica.

Como resultado, qualquer sistema capaz de efetuar comunicação utilizando os protocolos já bem estabelecidos da Internet, como citados acima, pode comunicar-se com um *Web service*. A única informação que o provedor do serviço e o consumidor precisam compartilhar são as entradas, saídas e sua localização, ou seja, disponibilizando sua interface e abstraindo totalmente a maneira como foram implementadas.

Segundo a visão apresentada em (W3C 2004b) a arquitetura dos *Web Services* está baseada na interação dos seguintes papéis: provedor do serviço, registrador de serviços e consumidor de serviços. As interações envolvem a publicação, a busca e a invocação do serviço. Neste esquema, o provedor do serviço desenvolve o serviço e publica sua descrição em um registrador de serviços, o consumidor de serviço efetua busca nos registros de serviços para descobrir o serviço que atende as suas necessidades, através das descrições adicionadas pelos provedores de serviços. Após descobrir o serviço ideal, o consumidor utiliza a descrição para interagir diretamente com o provedor e invocar o serviço desejado. Esta interação pode ser vista na figura 2.1.

Diante do exposto, algumas tecnologias foram desenvolvidas como forma a permitir tanto a publicação quanto a descoberta e invocação de serviços, como: registros UDDI (OASIS 2004), linguagem WSDL(W3C 2001) e o protocolo de troca de mensagens SOAP(W3C 2007).

Servidores de registro UDDI

Os registros UDDI[1] são repositórios de descrições de serviços que permitem a busca. Os consumidores de serviços podem efetuar busca pelos serviços nesses registros tanto em tempo de desenvolvimento, quanto em tempo de execução (*on-the-fly*), para recuperar as informações necessárias à invocação. A utilização dos registros UDDI para recuperação em tempo de desenvolvimento muitas vezes é desnecessária, já que o desenvolvedor pode obter essas infor-

[1] *Universal Description, Discovery and Integration*

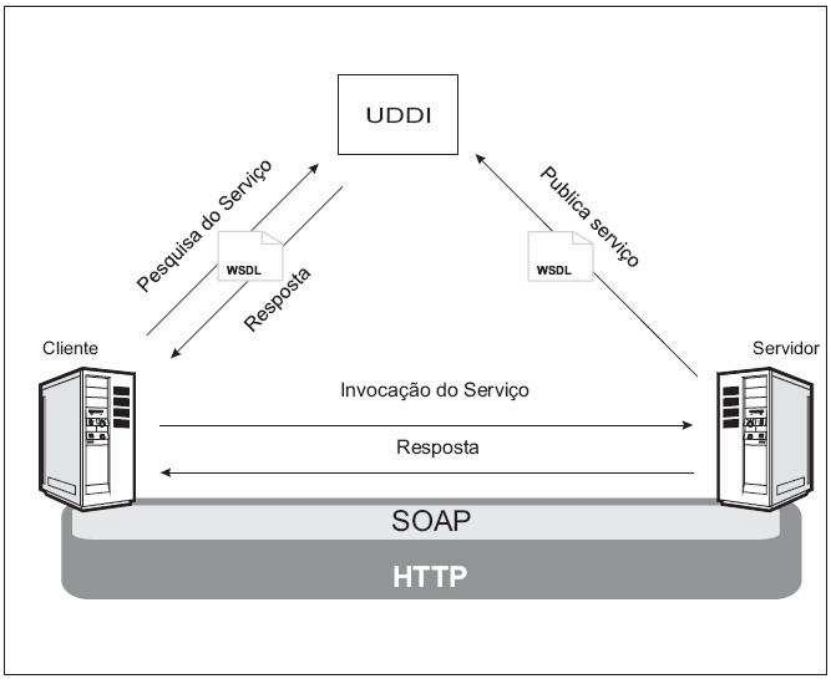

Figura 2.1: Modelo da interação entre cliente, servidor e registro UDDI

mações de outras fontes, como um site ou mesmo diretamente do provedor do serviço. A real serventia dos registros UDDI aparece quando o consumidor precisa descobrir um serviço em tempo de execução.

Apesar dos registros UDDI possibilitarem o armazenamento e a descoberta de novos serviços, sua especificação limita a forma como pode ser automatizado o processo de descoberta, já que sua estrutura está descrita meramente em termos sintáticos, não permitindo nenhuma inter-operação semântica, o que possibilitaria que todo o processo de descoberta, seleção e execução fosse feito de maneira automática.

Linguagem de descrição de *Web Services* – WSDL

WSDL é uma linguagem baseada em XML para descrição de interfaces, protocolos de invocação e detalhes de distribuição de *Web Services*. Um documento WSDL define serviços como coleções de pontos de acesso via rede chamados *ports*. A definicição de cada *Web Services* define sua localização, os parâmetros necessários e seus tipos, seu retorno, ou seja, todas as informações necessárias à invocação. WSDL complementa o modelo de representação de serviços utilizado pelos registros UDDI, ao prover uma descrição abstrata da interface do serviço e as formas de invocá-los (Zhou et al. 2006). Para tal, utiliza uma estrutura para descrição de serviços dividida em 6 elementos principais (W3C 2001): *Types*, *Message*, *PortType*, *Operation*, *Binding*, *Port* e *Service*.

Assim como a descrição provida pelos registros UDDI, a definição de serviços efetuada pelo WSDL, é puramente sintática, já que sua estrutura é baseada em estrutura de XML schema, não permitindo interoperação semântica em nível de ontologias.

2.1.2 *Semantic Web Services*

Os *Semantic Web Services* são a próxima etapa dos *Web Services*, pois no mundo da Web, em que a cada momento tenta-se agregar mais e mais conhecimento entendível por máquinas aos recursos, os *Semantic Web Services* podem ser vistos como um ponto de intersecção entre a tecnologia dos *Web Services* e a Web semântica(Berners-Lee et al. 2001) por prover a interoperabilidade semântica encontrada na Web semântica juntamente com a dinamicidade de recursos presentes nos *Web Services* (Daconta et al. 2003) (figura 2.2).

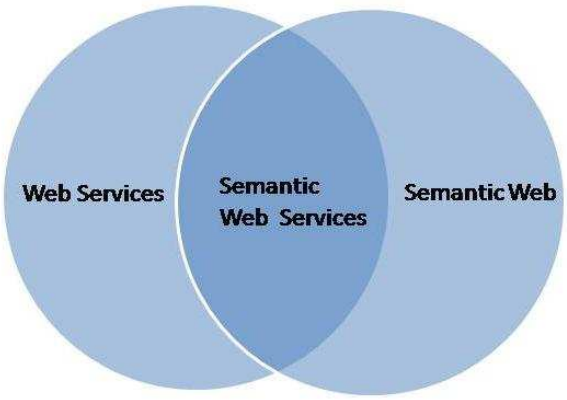

Figura 2.2: Visão Integrada de *Web Service* com *Semantic Web*

Segundo (Payne and Lassila 2004), os *Semantic Web Services* podem ser definidos como um aprimoramento das descrições dos *Web Services* através da utilização de anotações semânticas, de forma que possa ser conseguido um alto grau de automação, tanto na descoberta de serviços, quanto na composição, monitoramento e invocação de *Web Services*.

Para um melhor entendimento, imagine um cenário fictício em que um estudante de computação resolve tirar suas férias viajando para Europa. Para conseguir realizar tal tarefa ele terá de seguir um conjunto de passos como: fazer uma busca por preços de passagens aéreas, efetuar a reserva, confirmar o pagamento, fazer reservas em hotéis etc. Suponha que cada uma dessas tarefas possa ser satisfeita a partir da invocação de *Web Services* independentes. Utilizando *Web Services* tradicionais, em cada uma das etapas deveria haver

algum tipo de intervenção do usuário. Porém, se os *Web Services* estivessem anotados semanticamente, seria possível o estudante configurar um agente de software capaz de, sem intervenção humana, buscar e selecionar serviços que atendam aos requisitos do usuário, montar um plano de execução efetuando uma composição de todos os serviços selecionados e invocá-los de maneira automática.

Diante dos desafios de cenários como o colocado, diversas iniciativas de pesquisa estão sendo desenvolvidas com o intuito de agregar semântica à atual tecnologia de *Web Services*. Alguns modelos propõem um melhoramento das atuais linguagens de descrição de serviços, como o WSDL-S (W3C 2005), que neste caso teria como principal benefício a descoberta de serviços. Contudo, as principais iniciativas tem um objetivo mais ousado, que é prover uma maior semântica às descrições dos serviços, o que permitiria não apenas a descoberta automática, mas também a composição e a invocação automáticas de serviços.

De todas as iniciativas, as que mais se destacam por apresentarem modelos mais consistentes e completos são uma iniciativa européia, a WSML(Roman and et. al. 2008) [2](*Web Service Markup Language*) e uma iniciativa norte-americana, a OWL-S(W3C 2004a)[3]. Nessa disputa, a OWL-S leva, atualmente, uma vantagem em relação a sua concorrente, já que é recomendada pela W3C(W3C 2010), o que atrai um maior número de pesquisadores a trabalhar com ela.

2.1.3 OWL-S

Embora algumas vezes a OWL-S seja referida como uma linguagem, porque provê um vocabulário padrão para representação de serviços, trata-se de uma ontologia de topo (*Upper Ontology*) lançada em meados de 2004 em continuação ao projeto DAML-S (Ankolekar et al. 2005). Por ser uma recomendação da W3C, a OWL-S tem atraído o interesse de grande parte da comunidade científica relacionada a *Semantic Web Services* e grande parte das ferramentas desenvolvidas tem como base a utilização dela.

Motivação A OWL-S foi desenvolvida tendo como objetivo permitir a criação de *Semantic Web Services* capazes de atender a grande parte dos requisitos esperados de um serviço semântico (McILraith et al. 2001). Entre eles, os principais são:

[2]Anteriormente conhecida como WSMO (Web Service Modelling Ontology)
[3]*Ontology Web Languagen for Services*

- **Descoberta automática de serviços**: A OWL-S possibilita a descrição de serviços com um alto grau de semântica, o que permite que um agente de software descubra e selecione automaticamente um certo serviço de acordo com seu objetivo;
- **Invocação automática de serviços**: De forma semelhante ao item anterior, a OWL-S permite que os serviços possam ser invocados sem intervenção de nenhum agente humano;
- **Composição automática de serviços**: Dada a elevada semântica presente nas descrições de serviços, agentes de software estão aptos a realizar a combinação de serviços, sem qualquer intervenção humana, de maneira que uma tarefa complexa possa ser executada.

Estruturação Diferentemente da linguagem de descrição de serviços WSDL, a OWL-S efetiva a descrição de serviços semanticamente, o que permite uma maior interopeção. Segundo (Coalition 2003) a estruturação de uma ontologia de serviços é motivada pela necessidade de três essenciais tipos de conhecimento básicos sobre um serviço, o que poderia ser referenciado pela resposta às três perguntas seguintes:

- *O que o serviço faz?*
- *Como o serviço trabalha?*
- *Como acessar o serviço?*

A resposta às três perguntas acima permite que um agente de software saiba: qual a interface e o objetivo do serviço, quais são as etapas realizadas em sua execução e quais os protocolos necessários a sua invocação.

Diante do exposto, a OWL-S foi desenvolvida tendo em mente três componentes básicos:

1. Um perfil (*ServiceProfile*) que apresenta as "intenções" do serviço, isto é, qual a funcionalidade do serviço, o que ele provê;
2. Um modelo do Serviço (*ServiceModel*) que apresenta um maior detalhamento do funcionamento do serviço;
3. Um *ServiceGrounding*[4] que apresenta detalhes relacionado a maneira como o serviço pode ser invocado.

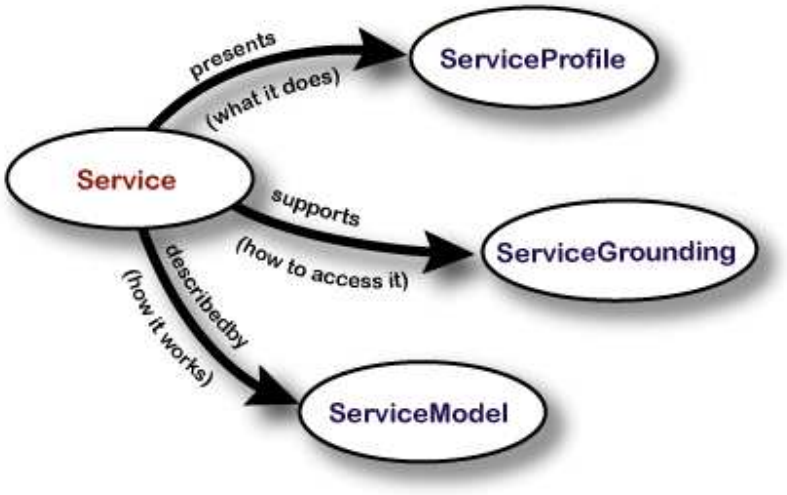

Figura 2.3: Estrutura proposta de um serviço semântico em OWL-S

O modelo abstrato de um *Semantic Web Services* em OWL-S com todos os relacionamentos entre as entidades presentes em um *Semantic Web Services* pode ser visto na figura 2.3.

A seguir será apresentada uma descrição mais detalhada da estrutura da OWL-S (perfil, modelo e Grounding), sendo dado maior enfoque ao perfil, por ter uma maior correlação com este trabalho.

ServiceProfile O objetivo do *ServiceProfile* é fornecer uma descrição do objetivo da utilização do Web service, de forma que o consumidor seja capaz de selecionar o serviço que mais atende as suas expectativas. Em geral as principais informações utilizadas são os tipos de entrada e retorno do serviço, pré-condições e efeitos da execução do serviço. A primeira vista, essas poucas informações não parecem suficientes para prover toda a interoperabilidade esperada de um serviço semântico, pois só se está utilizando tipos de entrada e saída para avaliações, no entanto, a interoperabilidade semântica é garantida pelo fato de que todos os parâmetros possuem ontologias que os representam no mundo. Suponha por exemplo dois serviços que efetuem a tarefa de um tradutor português-inglês, um sendo descrito em termos puramente sintáticos (*Web Services* tradicionais) e outro sendo descrito em OWL-S. No primeiro caso como a única informação disponível é a indicação que o serviço espera como entrada uma string e tem como retorno outra string, nenhum agente de software será capaz, de maneira automática, de perceber que se trata de um tradutor. Enquanto que, no caso do serviço descrito de maneira semântica,

[4]Obs: Não foi encontrado em nenhum texto, relacionado a OWL-S, em língua Portuguesa um termo equivalente, portanto, preferiu-se utilizar o termo no original em Inglês

há metadados relacionados a entrada e ao retorno (isto é, ontologias), que possivelmente indicaria que o texto de entrada é uma palavra em língua Portuguesa enquanto que a saída é em língua Inglesa, permitindo que agentes de software realizem raciocínios sobre tal conhecimento.

Diante deste contexto, o *Profile* foi desenvolvido para representar o primeiro passo no processo de utilização de um serviço semântico, que é a descoberta.

Nós podemos descrever o modelo do *Profile*, apresentado na Figura 2.4, através de três principais grupos :

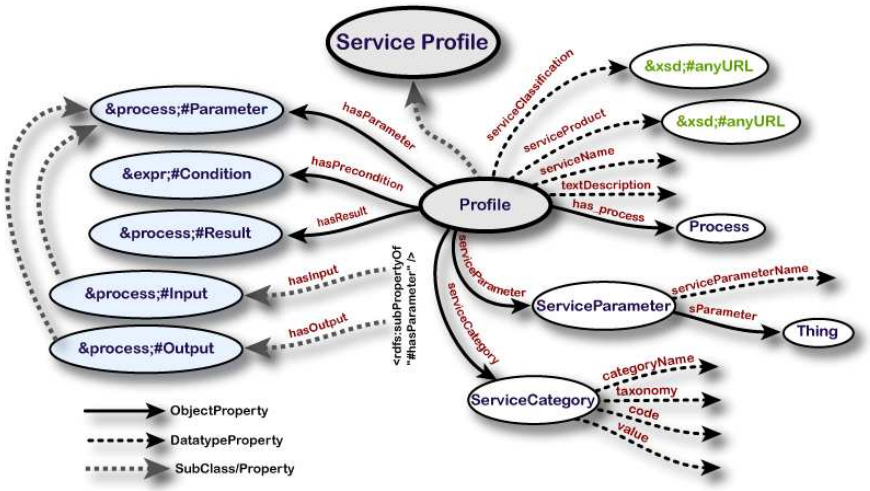

Figura 2.4: Classe Profile

O primeiro apresenta o nome e descrição.

- *serviceName* especifica o nome do serviço;
- *textDescription* dá uma descrição do serviço. Essa descrição é utilizada apenas para usuários humanos;

O segundo é um conjunto de descrições funcionais do serviço. Basicamente, apresenta as propriedades de entrada, saída, pré-condições e efeitos (IOPE's), sendo as principais propriedades que são utilizadas no processo de descoberta, nas ferramentas de busca. Os valores presentes nesses parâmetros são apontadores para classes presentes na ontologia *Process* representante do *ServiceModel* (Figura 2.3), de forma que se mantenha uma maior coesão na descrição do serviço;

- *hasParameter* é o conjunto de todas as entradas e saídas esperadas do serviço, aponta para uma instância da classe *Parameter* da ontologia *Process* ;

- *hasInput* especifica apenas as entradas que o serviço necessita para operar. Os tipos permitidos estão dentro do conjunto de instâncias da classe *Input* da ontologia *Process*
- *hasOutput* especifica todas as saídas que serão produzidas após a execução do serviço. Assim como a propriedade *hasInput*, os tipos aceitos seguem alguma instância presente na ontologia *Process*, mas nesse caso, da classe *Output*;
- *hasPrecondition* espefica as pré-condições que devem ser satisfeitas para que o serviço possa operar. Um exemplo de uma pré-condição seria um serviço de venda de livros, o usuário precisa de um cartão de crédito ainda válido. Pode variar de acordo com as instâncias definidas da classe *Condition*;
- *hasResult* especifica um dos resultados da execução do serviço, assim como definida pela classe *Result*. Ou seja especifica sobre quais condições a saída é gerada. Além disso, especifica quais serão as mudanças no domínio dada a execução do serviço.

O terceiro apresenta alguns atributos do *Profile*

- *serviceParameter* é uma lista de propriedades onde é possível especificar atributos de qualidade do serviço. Onde o valor da propriedade é uma instancia da classe *ServiceParameter* a qual possui dois atributos:
  - *serviceParameterName* é o nome do parâmetro
  - *sParameter* aponta para o valor do parâmetro dentro de uma ontologia OWL.
- *serviceCategory* que descreve a categoria do serviço. E possui quatro parâmetros: *categoryName* é o nome da catagoria, *taxonomy* armazena uma referência para o esquema de taxonomia, *value* aponta para o valor de um taxonomia específica e *code* cata tipo de serviço armazena o código associado para uma taxonomia.

Modelo do Serviço O objetivo do *ServiceModel* é dar uma visão bem mais detalhada do serviço em relação ao *ServiceProfile*. Este elemento da descrição terá sua utilização iniciada após o momento da descoberta e da seleção do serviço desejado. Isto é, quando o agente de software necessitar de informações mais concretas do funcionamento do serviço para poder efetuar interoperações (como composição de serviços para efetuar uma tarefa mais complexa, criação de um plano de execução, etc.), e posteriormente sua devida invocação, à partir do terceiro elemento serviço semântico que é o *Grounding*.

Para modelar esse conceito do serviço semântico, a OWL-S disponibiliza uma classe chamada (*Process*). Como já foi explicitado, muitas das propriedades presentes no *Profile* também apresenta-se no *ServiceModelo* como IOPE's. E à partir dessas propriedade que no *Process* é possível criar estruturas que descrevem como serviços diferentes podem ser composto para que uma tarefa complexa possa ser executada.

A classe *Process* não é utilizada de maneira direta, mas sim pela utilização de uma das três classes derivadas dela, que são:

1. **AtomicProcess**: É diretamente invocável (passando as devidas mensagens de entrada). O fluxo de execução de um processo atômico é análogo à de uma invocação de um método. Sendo assim, um processo atômico não possui nenhum sub-processo. Outra informação, que faz-se importante notar, é o fato que um processo atômico não apresenta o conceito de estado de execução, como ocorre com o serviço composto e portanto não efetua a monitoração durante a execução, pois não há o conceito de estado interno da execução;

2. **SimpleProcess**: Pode ser visto como uma abstração de um processo. Enquanto um *AtomicProcess* e um *CompositeProcess* tentam dar uma visão de uma caixa-de-vidro ao processo, um *SimpleProcess* se assemelha a uma caixa-preta. *SimpleProcess* são usados como elementos de abstração, no qual pode fornecer uma visão mais especializada de um *AtomicProcess* ou uma visão simplificada de um *compositeProcess* para efeitos de inferência e construção de planejamento. Um *SimpleProcess* se relaciona com os outros tipos de processo no fato de que um *AtomicProcess* realiza um *Simpleprocess* (pela propriedade *realize* e pela sua inversa, *realizedBy*) e um *compositeProcess* é uma expansão de um *SimpleProcess* (pela propriedade *expandTo*, havendo também sua propriedade inversa, *collapsesTo*);

3. **CompositeProcess**: São processos que podem ser decompostos em outros processo (atômicos ou complexos). Esses sub-processos serão executados de acordo com um conjunto de regras definidas em um plano de execução e na qual a entrada de dados para alimentar os sub-processos pode ser proveniente tanto do usuário do serviço, quanto de uma saída de um processo anterior.

Grounding O *Grounding* apresenta-se como último elemento a ser utilizado ao se trabalhar com *Semantic Web Services*, pois é justamente nele que está descrita a maneira como um serviço semântico pode ser invocado.

Ele pode ser considerado como o elemento mais próximo do mundo sintático se comparado ao perfil e ao modelo, já que ele pode ser visto como uma interface, ou "cola" entre as descrições puramente semânticas do perfil e do modelo e a descrição sintática do WSDL.

Desta forma, permitindo que o consumidor do serviço obtenha informações, como por exemplo, a maneira como serializar uma requisição, presente nos arquivos de descrição WSDL.

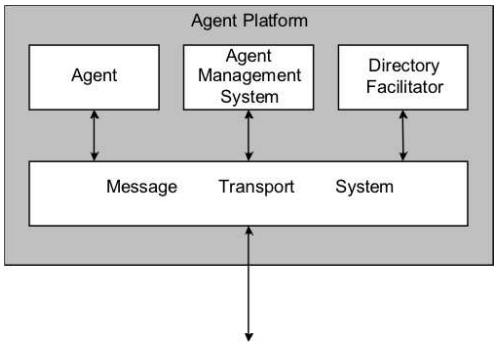

Figura 2.5: Referencia a arquitetura de uma plataforma de agente FIPA

2.2 JADE

JADE (Bellifemine et al. 2007) é um framework de desenvolvimento de software destinado a desenvolver sistemas multi-agente e aplicações conforme as normas FIPA(Odell and et. al. 2010) para agentes inteligentes. Inclui dois módulos principais: uma plataforma de agentes que segue o padrão FIPA e um pacote de classes para desenvolver agentes Java.

O modelo padrão de uma plataforma de agentes, tal como definido pela FIPA, é representado na figura 2.5.

O *Agent Management System* (AMS) é responsável pelo controle de acesso e a utilização da plataforma de agentes. Apenas um AMS existirá em uma única plataforma. O AMS possui um serviço de chamado de *white-page* que mantém um diretório de identificadores de agentes (AID) e um outro para controle dos diferentes estados que um agente pode exercer. Cada agente deve se registrar em uma AMS, a fim de obter um AID válido.

O *Directory Faciliatator* (DF) é responsável por fornecer o serviço chamado de *yellow page* o qual permite que agentes publiquem suas capacidades e que outros agentes possam descobrir tais capacidades.

O *Message Trasnport System*(MTS) é o componente que controla todas as trocas de mensagens dentro da plataforma, incluindo mensagens para/de plataformas remotas.

JADE cumpre com essa arquitetura de referência e, quando uma plataforma JADE é lançada, o AMS e DF são imediatamente criados e o módulo do MTS é definido para permitir a troca de mensagens.

Um agente deve ser capaz de realizar várias tarefas simultaneamente, em resposta a diferentes eventos externos. A fim de tornar a gestão eficiente, cada agente JADE é composto de uma única thread de execução e todas as suas tarefas são modeladas e podem ser implementadas como objetos *Behaviour*.

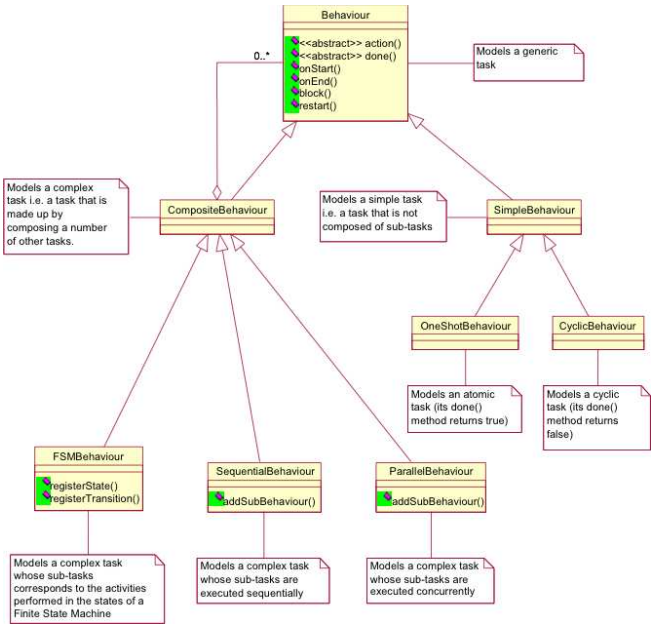

Figura 2.6: Comportamentos do JADE

A figura 2.6 apresenta um diagrama de classes UML dos comportamentos do JADE.

Behaviour Esta classe abstrata fornece uma base para modelar tarefas de agentes.

OneShortBehaviour Modela comportamentos atômicos que devem ser executados somente uma vez e não podem ser bloqueados;

CycliBehaviour Modela comportamentos atômicos que devem ser executados para sempre;

CompositeBehaviour Modela comportamento que são compostos de vários outros comportamentos;

SequentialBehaviour Executa seus *sub-behaviours* sequêncialmente e finaliza quando todos eles terminaram sua execução;

ParallelBehaviour Executa seus *sub-behaviours* paralelamente e finaliza quando uma particular condição presente *sub-behaviours* é encontrada;

FSMBehaviour Executa seus *sub-behaviours* de acordo uma Máquina de Estado Finita definida pelo usuário;

WakerBehaviour Implementa uma tarefa a ser executada somente uma vez após um período de tempo;

TickerBehaviour Implementa uma tarefa cíclica que deve ser executada periodicamente.

2.3 Raciocínio Baseado em Casos

O Raciocínio Baseado em Casos (RBC) resolve um problema atual baseando-se em experiências passadas. Tais experiências são mantidas em um repositório na estrutura de um par problema-solução. Onde cada par é conhecido como caso e o repositório como base de casos (BC). Quando um novo problema surge, algoritmos de similaridade são aplicados entre o problema e os casos na BC, utilizando apropriadas medidas de similaridades, a fim de encontrar os casos mais similares com o problema atual.

Por fim, as soluções dos casos mais similares são sugeridas como solução para o problema atual. Em outras palavras, a idéia básica em um RBC é que, para um domínio específico, os problemas a serem resolvidos tendem a ser recorrentes e repetir-se com pequenas alterações em relação a sua versão original. Desta forma, soluções anteriores podem ser aplicadas em problemas atuais com pequenas modificações, e muitos domínios reais têm adotado tal hipótese (Mantanas and et al. 2005).

O modelo clássico de RBC consiste de quatro fases que são coletivamente chamadas de 4R's, como apresentado na Figura 2.7, que são:

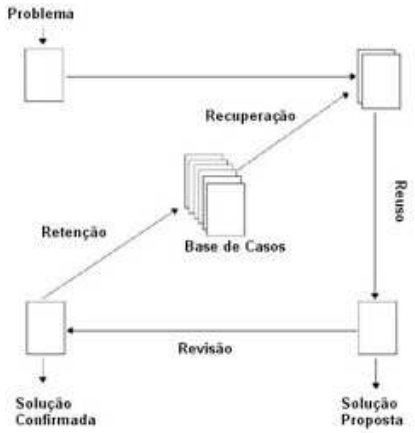

Figura 2.7: Ciclo RBC

- Recuperação: Nesta fase, os casos existentes na BC são recuperados e comparados com o caso atual.
- Reuso: As soluções dos casos mais similares são sugeridas como solução para o caso atual.
- Revisão: Em algumas situações é necessário adaptação da solução.
- Retenção: Uma vez que a solução é aplicada com sucesso, o par problema-solução (caso) deve ser armazenado na BC.

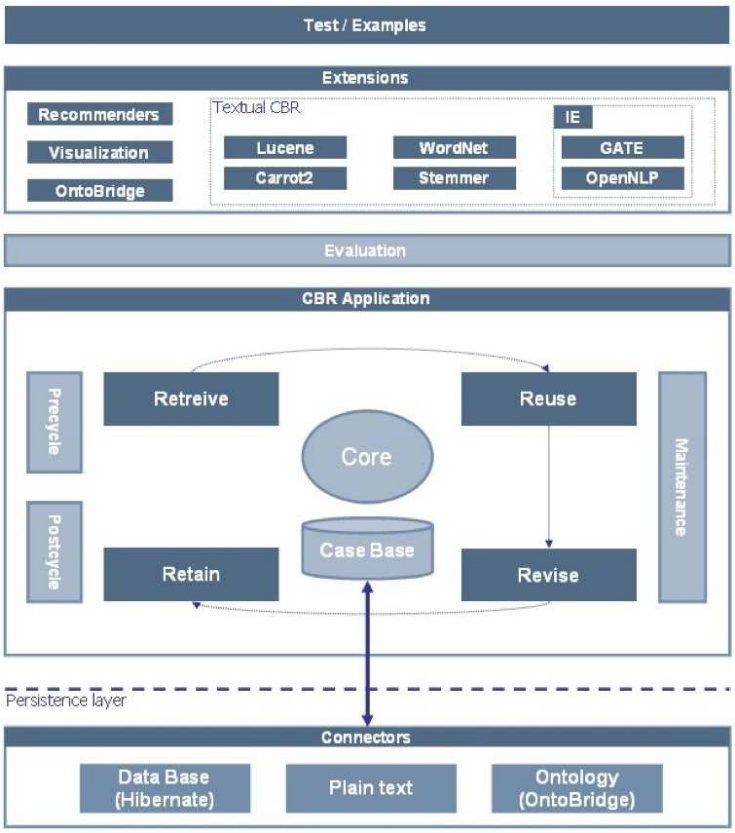

Figura 2.8: Arquitetura do Framework jColibri

2.4 Framework jColibri 2

jColibri2(Juan 2008) é um framework desenvolvido em Java para construir sistemas de CBR. Ele é resultado de experiências adquiridas da primeira versão do jColibri(Juan 2008). Ele resolve falhas da primeira versão, a exemplo representação de casos, gerenciamento de metadados, etc. A arquitetura do jColibri2 possui duas camadas: uma *white-box* que é orientada a programadores e outra *black-box* que é orientada a *designers*.

A Figura 2.8 apresenta a arquitetura jColibri2, juntamente, com os seus principais componentes:

Persistence Layer Inclui os conectores para armazenamento e recuperação de casos na camada de persistência. jColibri2 pode gerenciar qualquer *Data Base Manager* (DBM) pois ele utiliza internamente o hibernate(Bauer and King 2004) [5]. Em outras palavras, ao utilizar o Hibernate o jColibri2 permiti o uso de qualquer DBM, tais como Oracle

[5]Hibernate é um poderoso middleware que fornece serviços para persistência e *query* de dados

```
<DataBaseConfiguration>
    <HibernateConfigFile></HibernateConfigFile>
    <DescriptionMappingFile></DescriptionMappingFile>
    <DescriptionClassName></DescriptionClassName>
</DataBaseConfiguration>
```

Figura 2.9: *Arquivo databaseconfig.xml*

(Oracles 2010), MySQL(Oracle 2010), etc. Além disto, também fornece suporte a recuperação e armazenamento de informações em ontolgias. As configurações com o banco de dados são configuradas utilizando o arquivo *databaseconfig.xml*. Apresentado na Figura 2.9, onde as seguintes informações são definidas:

HibnernateConfigFile Define a localização do arquivo de configuração do Hibernate(Bauer and King 2004);

DescriptionMappingFile Define a localização do arquivo "hbm" do Hibernate;

DescriptionClassName Define a classe mapeada pelo arquivo "hbm" supracitado;

Core Este componente contém as classes e interfaces base do jColibri2.

Case-Base Define várias organizações em memória da base de casos.

Retrieve O mais importante método de recuperação é o *Nearest Neighbor*. Ele utiliza funções de similaridade local visando comparar simples atributos. Para tanto o jColibri2 disponibiliza as seguintes comparações: (Interval) calcula a distância entre dois números dentro de um intervalo (apresentada no Algoritmo 1), (Equal) retorna 1 se os dois indivíduos comparados são iguais, e zero caso contrário (apresentada no algoritmo 2) e (EnumDistance) retorna a distancia entre dois indivíduos numa lista ordenada (apresentada no algoritmo 3). E funções de similaridade global para comparar atributos compostos como apresentado no algoritmo 4. Embora este método seja o mais conhecido, o framework fornece outros, a exemplo *Expert Clerk Median Scoring* que recupera os casos de acordo com a mediana.

Algoritmo 1 computeInterval(double d1, double d2, int interval)

```
1: distance = Math.abs(d1 - d2)/interval
2: return 1 - distance
```

Algoritmo 2 computeEqual(Object o1, Object o2)

```
1: return o1.equals(o2) ? 1 : 0
```

Algoritmo 3 computeEnumDistance(Object o1, Object o2)

```
1: Enum e1 = (Enum)o1
2: Enum e2 = (Enum)o2
3: double size = e1.getDeclaringClass().getEnumConstants().length
4: diff = Math.abs(e1.ordinal() - e2.ordinal())
5: return 1 - (diff/size)
```

Algoritmo 4 computeSimilarityGlobal(double[] values, double[] weigths)

```
1: double acum = 0
2: double weigthsAcum = 0
3: for all int i=0; i < values.size(); i++) do
4:    acum += values[i] * weigths[i];
5:    weigthsAcum += weigths[i];
6: end for
7: return acum/weightsAcum
```

Reuse and Revise Estas duas fases são acopladas ao domínio específico da aplicação, assim jcolibri2 inclui apenas métodos simples para copiar a solução do caso para a consulta, para copiar apenas alguns atributos, ou para calcular a proporção direta entre a descrição e os atributos da solução.

Retain Armazena a solução sugerida a partir da aplicação do RBC.

Evaluation Métodos para medir a performance de uma aplicação RBC.

2.5 Algoritmo Genético

Algoritmo Genético (AG) constitui uma técnica de busca e otimização, inspirada no princípio Darwiniano de seleção natural e reprodução genética (Golderg 1989). Estes princípios são imitados na construção de algoritmos computacionais que buscam uma melhor solução para um determinado problema, através da evolução de populações de soluções codificadas através de cromossomos artificiais.

Em AG um cromossomo é uma estrutura de dados que representa uma das possíveis soluções do espaço de busca do problema. Cromossomos são então submetidos a um processo evolucionário que envolve avaliação, seleção, recombinação sexual (crossover) e mutação. Após vários ciclos de evolução a população deverá conter indivíduos mais aptos. Segundo (Pacheco 2009), podemos caracterizar os AG através dos seguintes componentes:

Problema a ser otimizado : AG são particularmente aplicados em problemas com diversos parâmetros ou características que precisam ser combinadas em busca da melhor solução; e problemas com grandes espaços de busca.

Representação das Soluções de Problema: A representação das possíveis soluções do espaço de busca de um problema define a estrutura do cromossomo a ser manipulado pelo algoritmo. Binária e números reais são as principais representações utilizadas.

Decodificação do Cromossomo: A decodificação do cromossomo consiste basicamente na construção da solução real do problema a partir do cromossomo.

Avaliação: A avaliação é o elo entre o AG e o mundo externo. A avaliação é feita através de uma função que melhor representa o problema e tem por objetivo fornecer uma medida de aptidão de cada indivíduo na população corrente, que irá dirigir o processo de busca. A função de avaliaç
ao é para um AG o que o meio ambiente é para seres humanos. Funções de avaliação são específicas de cada problema.

Seleção: O processo de seleção em AG seleciona indivíduos para a reprodução. A seleç ão é baseada na aptidão dos indivíduos: indivíduos mais aptos têm maior probabilidade de serem escolhidos para reprodução.

Operadores Genéticos: Indivíduos selecionados (e reproduzidos na população seguinte) são recombinados através de mutações e crossover.

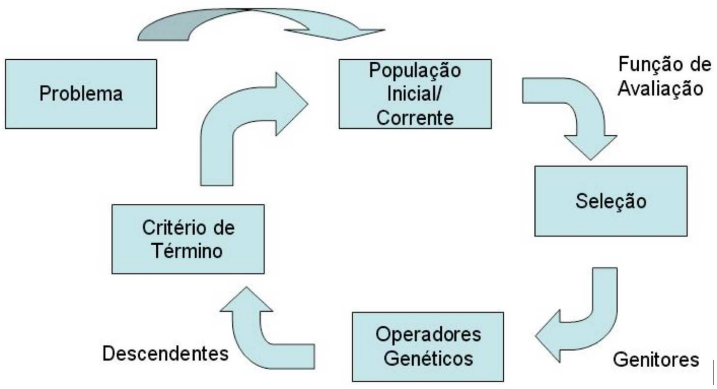

Figura 2.10: Ciclo do Algoritmo Genético

Critérios de Término: Em um algoritmo genético o tamanho da população, número de gerações e total de indivíduos são alguns dos parâmetros que podem ser utilizados como critério de parada.

Inicialização da População: A inicialização da população determina o processo de criação dos indivíduos para o primeiro ciclo do algoritmo. Tipicamente, a população inicial é formada a partir de indivíduos aleatoriamente criados. Populações iniciais aleatórias podem ser semeadas com bons cromossomos para uma evoluções mais rápida, quando se conhece, a priori, o valor de boas sementes.

Em suma, um algoritmo genético pode ser descrito como um processo contínuo que repete ciclos de evolução controlados por um critério de parada, conforme apresentado pela Figure 2.10.