4
Especificação e Implementação

4.1.
Objetivo

Esse capítulo tem como objetivo abordar questões relacionadas à especificação e à implementação da infra-estrutura proposta anteriormente. Seu foco é apresentar uma discussão sobre as principais funcionalidades mencionadas no capítulo anterior, procurando elucidar as decisões de projeto tomadas para resolvê-las.

Para tanto, serão apresentados vários diagramas de UML [Booch, 1998], principalmente o de classes e o de seqüência, visando a detalhar a arquitetura de *software* construída para suportar os requisitos levantados no Capítulo 3 para a infra-estrutura proposta. Também serão discutidos alguns detalhes sobre as ferramentas de terceiros que foram utilizadas na implementação do *framework* Matching Module e da OntoAPI.

PUC-Rio - Certificação Digital Nº 0310811/CA

4.2.
Framework Matching Module

4.2.1.
Diagrama de Classes

Devido à complexidade da classe de problemas que o *framework* visa atender, ele define um amplo conjunto de classes e de interfaces para lidar com as várias questões relacionadas a *matchmaking*. Procurou-se fazer um forte uso de soluções baseadas em *design patterns* [Gamma, 1995], uma vez que eles garantem flexibilidade, extensibilidade e modularidade ao projeto de *software*, além de constituírem um vocabulário de fácil entendimento sintático e semântico entre desenvolvedores.

A Figura 12 apresenta uma visão geral do diagrama de classes do *framework* Matching Module. No decorrer das próximas seções serão feitas discussões relacionadas a partes pontuais desse diagrama, o que permitirá um melhor entendimento de cada uma delas individualmente. As classes que estão destacadas correspondem a pontos de flexibilização, sendo esse detalhe mantido em todos os diagramas de classes apresentados nesse capítulo.

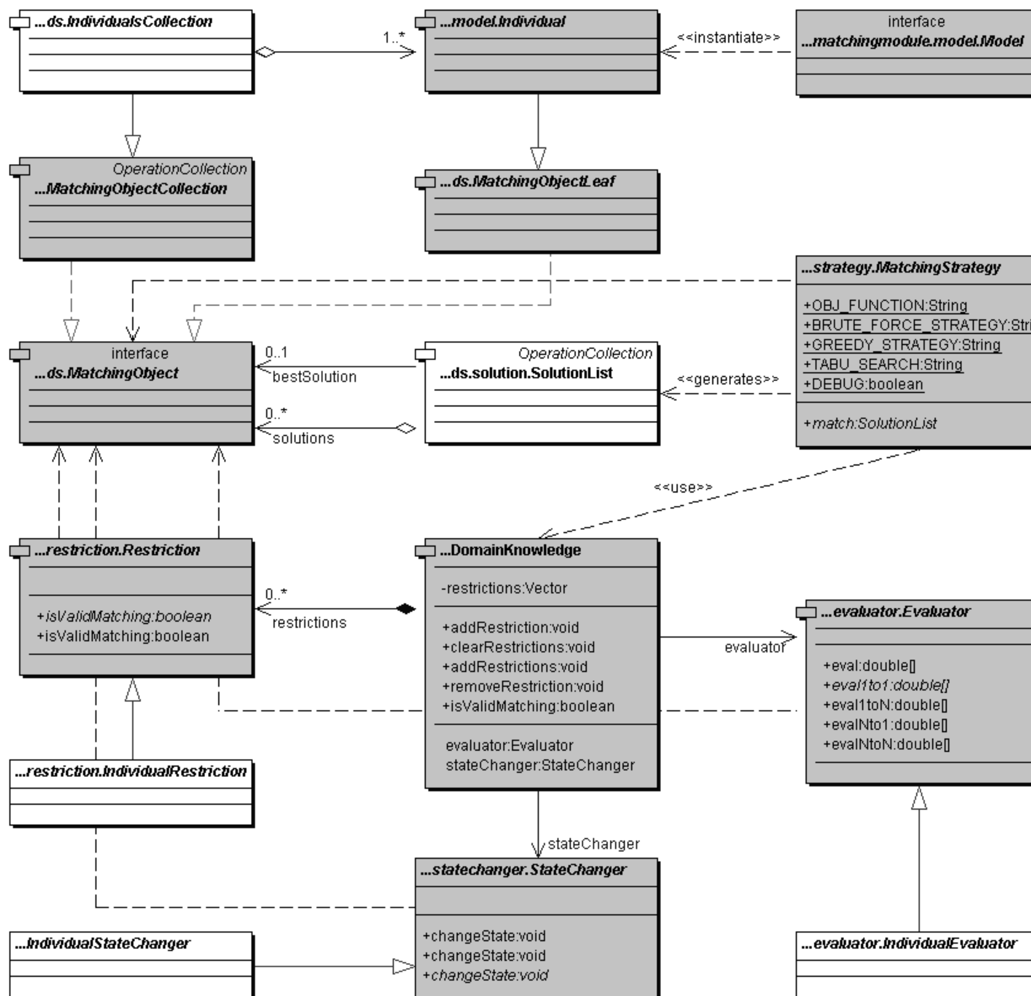

Figura 12 – Diagrama de classes do framework Matching Module

Nesse ponto, vale um pequeno comentário acerca da interface *MatchingObject* e de suas subclasses diretas: *MatchingObjectLeaf* e *MatchingObjectCollection*. Esse conjunto de classes representa um elo entre a parte do *framework* correspondente ao processamento do *matching* e aquela correspondente ao acesso ao modelo de dados, representada pelas classes *Model*, *Individual* e *IndividualsCollection*.

PUC-Rio - Certificação Digital Nº 0310811/CA

A razão pela escolha desse *design* de projeto foi baseada na expectativa de se criar mais um nível de abstração na representação das instâncias de um domínio particular, atendendo a situações que fogem o escopo desse trabalho. No contexto aqui discutido, todas as instâncias do domínio têm sua representação definida, exclusivamente, por objetos da interface *Individual*, acessadas via uma implementação particular da interface *Model*.

Uma vez que se fez a opção pela representação de um *MatchingObject*, todas as classes relacionadas ao processo de *matching*, tais como *MatchingStrategy*, *Restriction* e *Evaluator* foram especificadas segundo essa interface. Para evitar o uso explícito, e constante, de conversões de tipos para a interface *Individual*, o *framework* disponibiliza classes tais como *IndividualRestriction*, *IndividualEvaluator* e *IndividualStateChanger*, que definem especializações das classes anteriores em que suas interfaces de métodos estão baseadas no tipo *Individual*.

PUC-Rio - Certificação Digital Nº 0310811/CA

4.2.2. Domain Knowledge

Todas as classes responsáveis por modelar as informações dependentes de domínio, necessárias para o processo de *matchmaking*, estão apresentadas no diagrama de classes da Figura 13. A classe abstrata *DomainKnowledge* representa um agregador de informações acerca do domínio em questão. Dessa forma, ela mantém referências para um *Evaluator*, um *StateChanger* e um conjunto de objetos do tipo *Restriction*.

A classe abstrata *Evaluator* foi projetada segundo o padrão de projeto *Strategy*. Todas as classes que estendem um *Evaluator* devem definir uma implementação para o método abstrato *eval1to1*, que é utilizado para calcular o grau de similaridade entre duas instâncias de um dado domínio. São disponibilizadas implementações padrão para os demais métodos de avaliação de similaridade, que devem ser sobrescritos caso seja necessário executar outros tipos de *matching* para o domínio, que não seja apenas entre instâncias simples.

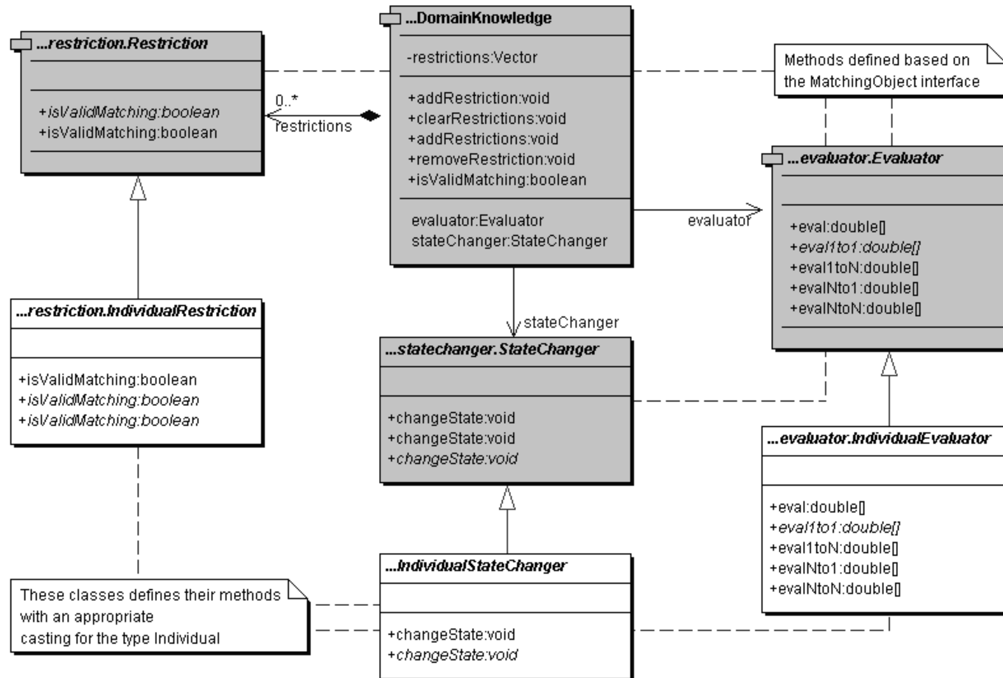

Figura 13 – Diagrama de classes relacionado ao Domain Knowledge

A classe abstrata *StateChanger* segue um projeto similar. Todas as suas extensões devem implementar um método abstrato *changeState*. Esse método define como o estado de uma instância do domínio deve ser alterado quando uma escolha é feita na construção de uma solução de *matching*.

A classe abstrata *Restriction* também segue o padrão *Strategy*. Uma extensão dessa classe deve implementar o método abstrato *isValidMatching*, que verifica a validade de uma possível escolha de *matching* com relação à restrição em questão. O *framework* disponibiliza alguns tipos de restrições já definidos, que podem ser utilizados ou estendidos de acordo com a necessidade de um domínio qualquer.

A classe *DomainKnowledge* apresenta métodos para manipulação dos objetos *StateChanger* e *Evaluator* associados, bem como para inclusão de novos objetos do tipo *Restriction*. Ela também define um método *isValidMatching*, que tem como comportamento executar todas as avaliações definidas para cada uma das restrições associadas.

PUC-Rio - Certificação Digital Nº 0310811/CA

4.2.3. Model

O diagrama de classes da Figura 14 apresenta o conjunto de classes do *framework* responsáveis por representar um modelo de dados. A interface *Model* tem como finalidade definir um conjunto de métodos responsáveis por acessar e obter informações sobre as entidades do domínio, que são representadas por objetos do tipo *Individual*.

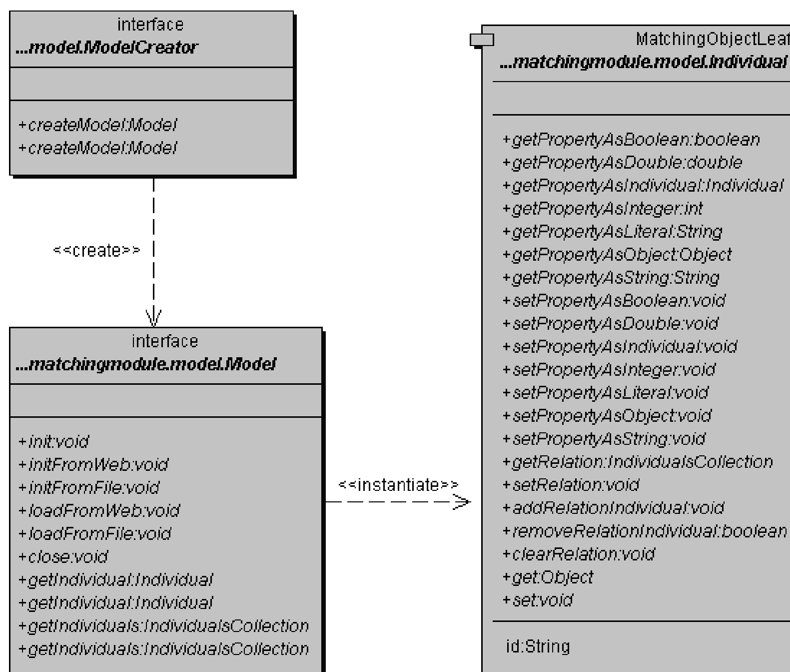

Figura 14 – Diagrama de classes relacionado ao Model

A interface *Individual* disponibiliza vários métodos para obter as informações acerca das propriedades das entidades do domínio que seus objetos referenciam. Existem métodos que permitem obter os valores das propriedades cuja imagem seja tipos primitivos, tais como *strings*, números inteiros, dentre outros.

As propriedades, cujos valores de imagem são outros objetos do tipo *Individual*, são tratadas de forma diferenciada dos tipos primitivos. Essas propriedades são consideradas como relações, obtidas através da execução do método *getRelation*, cujos valores são sempre representados por objetos da classe *IndividualsCollection*. São também disponibilizados métodos para manipulação do valor dessas relações, permitindo a inclusão e a remoção de

PUC-Rio - Certificação Digital Nº 0310811/CA

PUC-Rio - Certificação Digital Nº 0310811/CA

objetos. Porém, essas alterações só são realizadas em nível de memória, não sendo refletidas de forma persistente no modelo de dados original.

Para cada *Model* deve ser definido um *ModelCreator*, que define uma interface para criação de novos objetos que tratam da gerência do modelo de dados. Seu comportamento é baseado no padrão de projeto *Factory Method*, uma vez que cada uma de suas extensões deve implementar o método *createModel*, que permite gerar uma nova instância de um objeto *Model*. A classe *ModelCreator* também define uma segunda versão desse método, em que é possível definir um conjunto de parâmetros que podem ser utilizados na construção de um objeto *Model*.

4.2.4. Matching Strategy

A Figura 15 apresenta o diagrama de classes relacionado às estratégias de *matching* utilizadas pelo *framework* Matching Module.

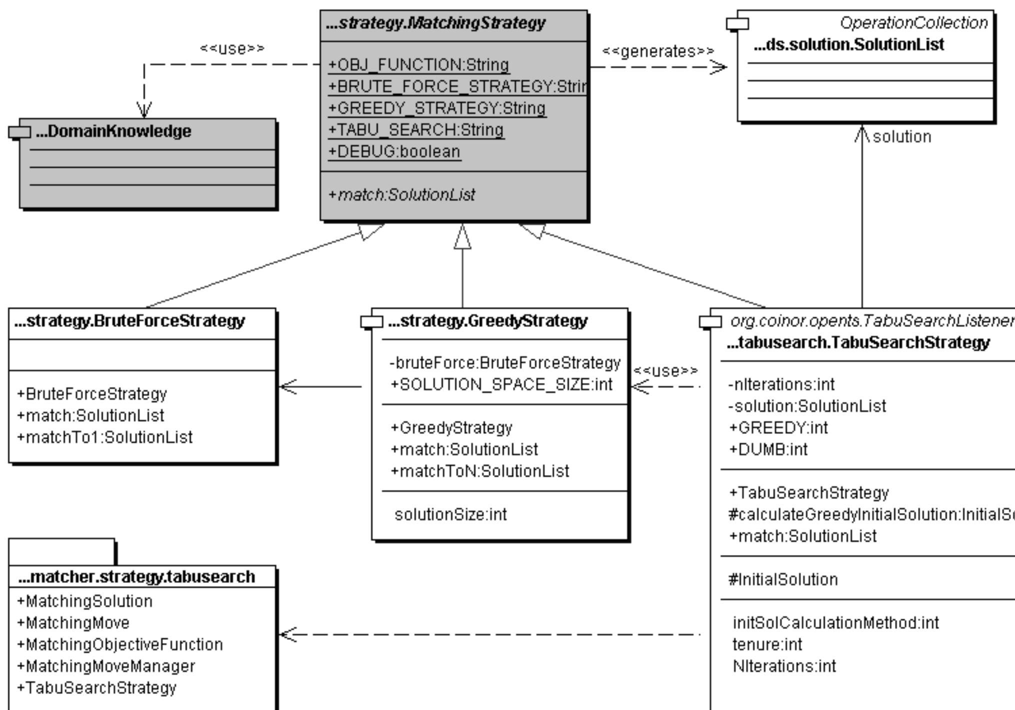

Figura 15 – Diagrama de classes relacionado à Matching Strategy

A classe abstrata *MatchingStrategy* foi projetada segundo o padrão de projeto *Strategy*. Novas extensões dessa classe podem ser implementadas,

sendo definidas para cada uma delas diferentes estratégias de execução de *matching*. O método abstrato *match* define o comportamento sendo implementado por uma estratégia particular.

A Figura 16 descreve um trecho de código que ilustra o relacionamento entre uma estratégia de *matching* e os objetos que representam informação dependente de domínio: uma função de similaridade, definida em um *Evaluator*, e um método de validação da escolha de *matching*, definida por implementações da classe *Restriction*. O código apresentado foi retirado da implementação referente à estratégia *BruteForceStrategy*, sendo parte integrante do *framework*.

```
public class BruteForceStrategy extends MatchingStrategy
{
    ...

    public SolutionList matchTo1( MatchingObject i1,
                                  MatchingObject i2,
                                  DomainKnowledge domainKnowledge,
                                  boolean maximize )
            throws MatchingException
    {
    //  Code for initialization
        ...
        Iterator solSpace = ((MatchingObjectCollection)i2).iterator();
        while( solSpace.hasNext() )
        {
        //  Iterate through the possible solutions, looking for the best

            MatchingObject currSolution = (MatchingObject) solSpace.next();

        //  Check to see if the current solution is a valid one
            if( !domainKnowledge.isValidMatching( i1, currSolution ) )
            {
                continue;
            }

        //  Calculate its objective function value
            objFunction = evaluator.eval( i1, currSolution );
            if( objFunction != null )
            {
                solutions.addValue( currSolution, OBJ_FUNCTION, objFunction );
            }

        //  Compare the current solution which the others solutions
            ...

            solutions.addSolution( currSolution );
        }

        return solutions;
    }
}
```

Figura 16 – Relacionamento entre uma Matching Strategy e informação de domínio

O comportamento de uma busca baseada em força bruta pode ser observado pela iteração sobre todos as instâncias do domínio que compõem o espaço de possíveis soluções para o problema de *matchmaking*. Pode-se perceber que se o *matching* não for válido, o valor de similaridade não será calculado. Caso contrário, será feita a análise de similaridade entre a demanda do domínio e a instância sendo testada.

PUC-Rio - Certificação Digital Nº 0310811/CA

A estratégia *TabuSearchStrategy* foi implementada através do *framework* OpenTS [OpenTS], que corresponde a um conjunto de classes Java que permitem implementar uma meta-heurística de busca tabu, baseado num modelo de orientação a objetos. Esse *framework* tem como características ser independente do problema a ser resolvido, esconder grande parte do trabalho relacionado à execução do algoritmo de busca tabu, definir uma estrutura logicamente mais organizada para representação do problema, dentre outros.

A Figura 17 apresenta o ciclo de iteração no *framework* OpenTS. Um componente do *framework* propõe mudanças nos itens que compõem uma dada solução. Cada nova solução, decorrente das alterações anteriores, é avaliada. Caso alguma dessas soluções seja melhor que a inicial, então ela é definida como nova solução e o processo entra num novo ciclo.

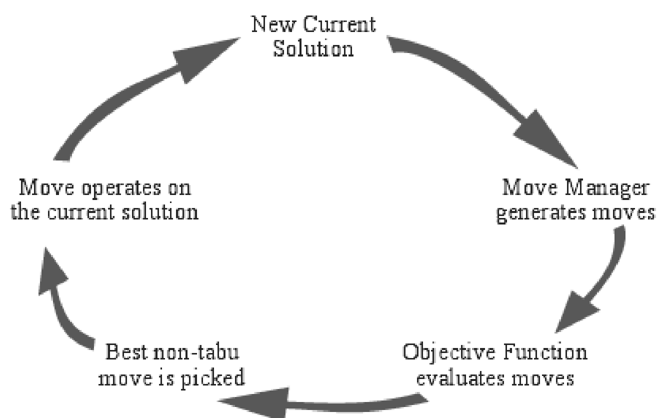

Figura 17 – Ciclo de iteração do framework OpenTS

PUC-Rio - Certificação Digital Nº 0310811/CA

4.2.5. Instanciando um Domínio de Conhecimento

Essa seção exemplifica como instanciar o *framework* Matching Module para trabalhar num domínio de conhecimento particular. Para tanto, serão apresentadas implementações das classes *DomainKnowledge*, *Evaluator*, *StateChanger* e *Restriction* para o domínio de gerência de competências, em que se deseja propor cursos para pessoas.

As implementações apresentadas nessa seção são base para os exemplos de uso da infra-estrutura proposta, que serão discutidos no Capítulo 5. Não existe, nesse momento, preocupação em apresentar o esquema de estruturação da informação das instâncias do domínio em questão, ou seja, das pessoas, das competências e dos cursos. Dessa forma, caso haja interesse por parte do leitor em conhecer previamente o modelo da informação aqui utilizado, deve-se ler a seção relacionada no capítulo posterior.

Como as implementações de *DomainKnowledge* têm com função encapsular as definições das demais classes citadas anteriormente, a implementação particular para o domínio em questão será apresentada apenas no final da seção. O primeiro passo será apresentar a implementação relacionada ao *Evaluator*. Porém, nesse momento, ela apenas atenderá ao caso mais simples, em que se deseja avaliar a similaridade entre uma pessoa e um curso. A Figura 18 apresenta a declaração para a classe *PersonCourseEvaluator*.

PUC-Rio - Certificação Digital Nº 0310811/CA

```
public class PersonCourseEvaluator extends IndividualEvaluator
{
}
```

Figura 18 – Declaração da classe PersonCoursEvaluator

Para especificar a forma como será avaliada a similaridade entre as instâncias do domínio, para o caso *1to1*, deve-se implementar o método *eval1to1* do *Evaluator*. A Figura 19 apresenta uma possível maneira de se calcular tal similaridade.

Para a implementação em questão, a interpretação de similaridade está relacionada ao maior número de competências atendidas, ou seja, supondo que uma pessoa requer determinadas competências e que um curso desenvolve determinadas competências, a similaridade é proporcional ao número de competências desenvolvidas pelo curso que atendem aquelas requeridas pela pessoa em questão.

```
public class PersonCourseEvaluator extends IndividualEvaluator
{
    public double[] eval1to1( Individual person, Individual course )
    {
        float count = 0.0f;

        if( hasCoursed( person, course ) )
        {
            return new double[] {-100.0f};
        }
        if( !checkForPreRequisites( person, course ) )
        {
            return new double[] {-50.0f};
        }

        try
        {
            Iterator requires = person.getRelation( "requires" ).iterator();
            while( requires.hasNext() )
            {
                Individual r = (Individual) requires.next();

                Iterator develops = course.getRelation( "develops" ).iterator();
                while( develops.hasNext() )
                {
                    Individual d = (Individual) develops.next();

                    Comparator cmp = new StringComparator();
                    if( cmp.compare( r.getId(), d.getId() ) == 0 )
                    {
                        count++;
                    }
                }
            }
        }
        catch( Exception e )
        {
        }
        return new double[] {(float)count};
    }
}
```

Figura 19 – Implementação da função eval1to1 da classe PersonCourseEvaluator

No código acima, a função recebe dois objetos da classe *Individual*, que correspondem à pessoa e ao curso. Primeiramente, é verificado se a pessoa já fez o curso que está sendo proposto e se todos os cursos feitos anteriormente por ela satisfazem os pré-requisitos do mesmo. As competências requeridas pela pessoa e desenvolvidas pelo curso são obtidas via a o método *getRelation*, que retorna um conjunto de instâncias do domínio que está associado, segundo alguma propriedade, a uma instância particular. Nesse caso, esse conjunto

PUC-Rio - Certificação Digital Nº 0310811/CA

corresponde às competências requeridas pela pessoa e desenvolvidas pelo curso.

Um *StateChanger* representa o elemento de informação do domínio de conhecimento responsável por especificar como ocorre a mudança de estado de uma instância desse domínio nos casos em que a construção da solução de *matching* é realizada de forma incremental. Para que se possam propor seqüências de cursos para pessoas, faz-se necessário, então, escrever uma implementação dessa classe.

```
public class PersonCourseStateChanger extends IndividualStateChanger
{
    public void changeState( Individual person, Individual course )
            throws StateChangeException
    {
        if( ( person == null ) || ( course == null ) )
        {
            return;
        }
        try
        {
            person.addRelationIndividual( "hasCoursed", course );
            IndividualsCollection develops = course.getRelation( "develops" );
            if( develops != null )
            {
                Iterator iter = develops.iterator();
                while( iter.hasNext() )
                {
                    Individual currComp = (Individual) iter.next();
                    person.removeRelationIndividual( "requires", currComp );
                }
            }
        }
        catch( Exception e )
        {
            throw new StateChangeException();
        }
    }
}
```

Figura 20 – Implementação da classe PersonCourseStateChanger

A mudança de estado especificada pela implementação da Figura 20 define que, dados uma pessoa e um curso, deve-se adicionar o curso na relação *hasCoursed* da pessoa, bem como remover todas as competências requeridas pela pessoa que são desenvolvidas pelo curso em questão.

PUC-Rio - Certificação Digital Nº 0310811/CA

Com a implementação da classe *PersonCourseStateChanger*, passa a ser possível definir o caso de *matching 1toN*. O trecho de código da Figura 21 apresenta a implementação da função *eval1toN* da classe *PersonCourseEvaluator*, que define como avaliar a similaridade entre uma pessoa e um conjunto de cursos.

```
public class PersonCourseEvaluator extends IndividualEvaluator
{
    public double[] eval1toN(Individual person, IndividualsCollection courses)
    {
        double result = 0.0f;
        double firstValue = -1.0f;
        boolean firstTime = true;
        double totalPrice = 0.0f;
        try
        {
            Individual p = (Individual) person.getClone();
            Iterator iter = courses.iterator();
            while( iter.hasNext() )
            {
                Individual currCourse = (Individual) iter.next();
                double[] currEval = eval1to1( p, currCourse );
                if( currEval[0] >= 0 )
                {
                    stateChanger.changeState( p, currCourse );
                }
                if( currEval[0] == 0 )
                {
                    currEval[0] = -0.1f;
                }
                result += currEval[0];
                totalPrice += currCourse.getPropertyAsDouble( "price" );
                if( firstTime )
                {
                    firstValue  = currEval[0];
                    firstTime = false;
                }
            }
            return new double [] { result, -totalPrice, firstValue };
        }
        catch( Exception e )
        {
        }
        return new double[] { result };
    }
}
```

Figura 21 – Implementação da função eval1toN da classe PersonCourseEvaluator

Como pode ser notado no código acima, o algoritmo de avaliação do caso *1toN* é reduzido a *N* aplicações do método de avaliação do caso *1to1*. Para cada curso da seqüência, se ele pode ser feito pela pessoa então é aplicada a

PUC-Rio - Certificação Digital Nº 0310811/CA

operação de mudança de estado definido pela implementação do *StateChanger*. Perceba que mesmo que o curso não desenvolva nenhuma das competências requeridas pela pessoa, ele entra na solução. Porém, nesse caso, o resultado da primeira dimensão do valor de similaridade sofre uma certa penalidade. As outras dimensões do resultado de *matching* dizem respeito, ordenadamente, ao custo total para fazer a seqüência de cursos proposta e ao número de competências atendidas pelo primeiro curso da seqüência.

Um outro aspecto importante do código acima é o uso da função *getClone* invocada pelo objeto *person*. Essa função permite gerar uma instância idêntica de um objeto *Individual*. Essa clonagem é necessária pois durante o processo de *matchmaking* várias soluções de *matching* serão propostas para a pessoa e, para a construção de cada uma delas, o estado da pessoa é alterado. Se não houvesse essa clonagem, a segunda solução a ser construída levaria em consideração o estado da pessoa definido após a construção da primeira solução. Dessa forma, a clonagem permite resguardar o estado original da pessoa, para que cada construção de solução do *matching* possa iniciar com as informações corretas.

Um outro elemento importante relacionado à instanciação de um domínio de conhecimento corresponde às restrições que serão aplicadas para analisar se uma solução de *matching* é válida. Para o domínio sendo analisado, foi definida apenas uma restrição, que está descrita no trecho de código da Figura 22, correspondente à implementação da classe *PersonCourseRestriction*.

A restrição implementada é bem simples, tendo como comportamento invalidar soluções de *matching* em que o curso proposto ou que algum dos cursos propostos já tenham sido feitos anteriormente pela pessoa.

PUC-Rio - Certificação Digital Nº 0310811/CA

```
Public class PersonCourseRestriction extends IndividualRestriction
{
    public boolean isValidMatching( IndividualsCollection origin,
                                    Individual solution )
            throws RestrictionException
    {
        Iterator it = origin.iterator();
        while( it.hasNext() )
        {
            if( !isValidMatching( (Individual) it.next(), solution ) )
            {
                return false;
            }
        }
        return true;
    }

    public boolean isValidMatching( Individual origin, Individual solution )
            throws RestrictionException
    {
        try
        {
            return !evaluator.hasCoursed( origin, solution );
        }
        catch( Exception e )
        {
            return false;
        }
    }
}
```

Figura 22 – Implementação da classe PersonCourseRestriction

A última tarefa a ser executada, para que se tenha um domínio de conhecimento pronto para ser utilizado para resolver problemas de *matching*, é a implementação da classe relacionada ao domínio de conhecimento. A classe *PersonCourseDomainKnowledge* corresponde à implementação para o domínio relacionado à gerência de competências. O trecho de código da Figura 23 apresenta a sua definição.

```
public class PersonCourseDomainKnowledge extends DomainKnowledge
{
    public PersonCourseDomainKnowledge()
    {
        super( new PersonCourseEvaluator(), new PersonCourseStateChanger() );
        addRestriction( new PersonCourseRestriction() );
    }
}
```

Figura 23 – Implementação da classe PersonCourseDomainKnowledge

PUC-Rio - Certificação Digital Nº 0310811/CA

4.2.6. O Arquivo de Configuração do Framework Matching Module

Visando a facilitar o gerenciamento dos seus *hotspots*, o *framework* Matching Module disponibiliza um arquivo de configuração. Nesse arquivo, podem ser definidos todos os domínios de conhecimento instanciados, os modelos de dados suportados, bem como as estratégias de *matching* que podem ser utilizadas para resolver os problemas de *matchmaking*. A Figura 24 apresenta um exemplo de tal arquivo.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<matchingmodule-config>

    <!—- List of Model implementations -->
    <models>
        <model name="ONTOLOGY">
            <creator class="full_package_path.OntologyModelCreator"/>
            <param name="CONFIG_FILE" value="ontoapi-config.xml"/>
        </model>
    </models>

    <!- List of DomainKnowledge implementations -->
    <domain-knowledges>
        <domain-knowledge
            name="PersonCourseDomainKnowledge"
            class="full_package_path.PersonCourseDomainKnowledge"/>
    </domain-knowledges>

    <!-- List of MatchingStrategy implementations -->
    <matching-strategies>
        <matching-strategy
            name="Brute Force"
            class="full_package_path.BruteForceStrategy"/>
        <matching-strategy
            name="Greedy"
            class="full_package_path.GreedyStrategy"/>
        <matching-strategy
            name="Tabu Search"
            class="full_package_path.TabuSearchStrategy"/>
    </matching-strategies>

</matchingmodule-config>
```

Figura 24 – Arquivo de configuração do framework Matching Module

Os nomes completos dos pacotes foram omitidos para tornar menos confusa a visualização do conteúdo do arquivo. O *framework* possui um componente que lê o arquivo de configuração gerando instâncias de cada uma das classes especificadas, utilizando para isso o suporte a reflexão, disponibilizado pela linguagem Java (pacote *java.lang.reflect*).

PUC-Rio - Certificação Digital Nº 0310811/CA

4.2.7.
A Camada de Serviços do Framework Matching Module

A camada de serviços do *framework* disponibiliza uma interface para que outras aplicações possam fazer uso das funcionalidades de *matching*, definidas para todos os domínios de conhecimento cadastrados, ou seja, que foram especificados no arquivo de configuração do *framework*.

Essa camada foi implementada segundo o paradigma de *Web Services*, utilizando como infra-estrutura o Apache Axis [Axis]. O Axis é um *framework* de suporte para comunicação e desenvolvimento de aplicações que precisam lidar com o protocolo SOAP. O Jakarta Tomcat [Tomcat] foi utilizado como *container web*, servindo de base para a execução do Axis e, conseqüentemente, dos serviços de *matching* que foram definidos.

A Figura 25 apresenta uma visão geral dos componentes de *software* que compõem a camada de serviços.

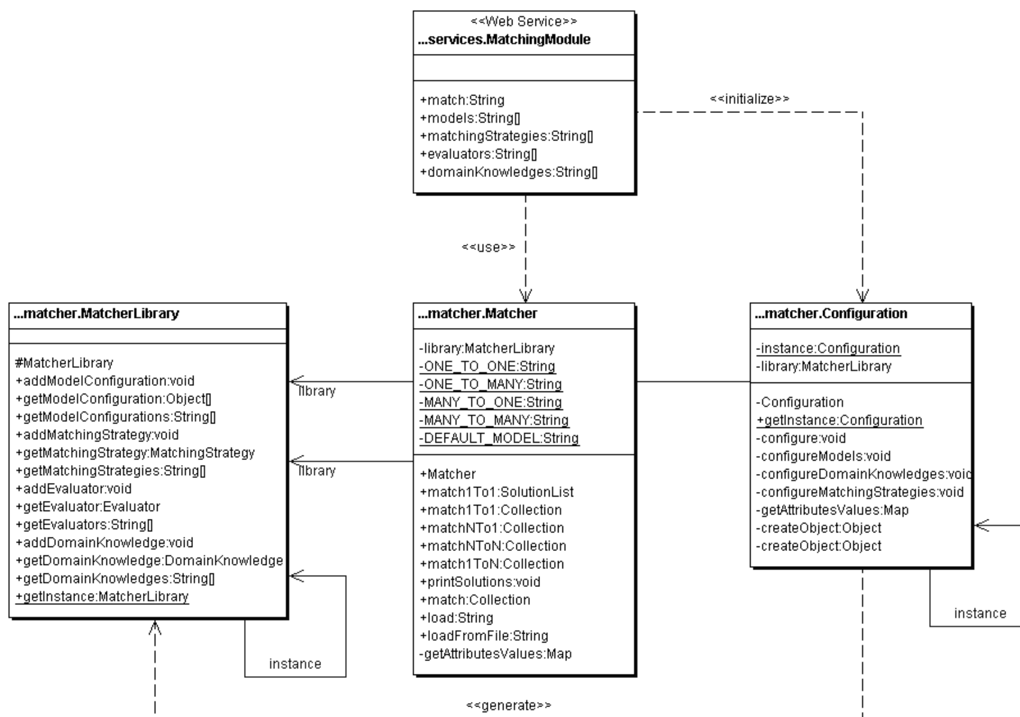

Figura 25 – Diagrama de classes da camada de serviços

PUC-Rio - Certificação Digital Nº 0310811/CA

A classe *MatchingModule* define os serviços disponibilizados. No momento de sua criação, ela inicializa a classe *Configuration*, que é responsável pela leitura do arquivo de configuração do *framework*. Durante a leitura desse arquivo, a classe *MatcherLibrary* é preenchida com os objetos apropriados, instanciados a partir das informações constantes no arquivo de configuração.

Todas as operações relacionadas à requisição de informações acerca dos domínios de conhecimento, modelos de dados e estratégias de *matching* cadastradas para uso são obtidas através das informações contidas na classe *MatcherLibrary*. A operação de *matching*, em particular, é executada via um objeto do tipo *Matcher*. Esse componente sabe, a partir de uma requisição de *matching*, criar uma instância dinâmica do *framework* capaz de executar o processo de *matchmaking* descrito pelos parâmetros dessa requisição.

Uma requisição de *matching* é definida via um documento XML, que é construído por uma entidade externa e passado para um *Matcher*. A Figura 26 apresenta um exemplo de uma requisição em que é definido um problema de *matching* do tipo *1to1*, sobre o domínio de gerência de competências, que utiliza a estratégia *BruteForceStrategy* e cujos dados das instâncias estão descritas utilizando-se ontologias.

PUC-Rio - Certificação Digital Nº 0310811/CA

```
<?xml version="1.0"?>
<Matchings>
    <Matching type="1to1" model="ONTOLOGY" name="example">
        <Problem>
            <Elements>
                <Element
id="http://www.lac.inf.puc-rio.br/~ferrao/ontologies/rdf/people/luis.rdf" />
            </Elements>
        </Problem>
        <SolutionSpace>
            <Elements>
                <Element
id="http://www.lac.inf.puc-rio.br/~ferrao/ontologies/rdf/courses/course1.rdf" />
                <Element
id="http://www.lac.inf.puc-rio.br/~ferrao/ontologies/rdf/courses/course2.rdf" />
                <Element
id="http://www.lac.inf.puc-rio.br/~ferrao/ontologies/rdf/courses/course3.rdf" />
            </Elements>
        </SolutionSpace>
        <Strategies>
            <BruteForceStrategy name="brute_force" maximize="true">
                <DomainKnowledge name="PersonCourseDomainKnowledge" />
            </BruteForceStrategy>
        </Strategies>
    </Matching>
```

```
</Matchings>
```

Figura 26 – Exemplo de uma requisição de matching

A *tag Matching* define o tipo de problema a ser executado, nesse caso um *matching 1to1*, o nome de referência para o problema em questão, *example* para essa requisição, bem como o tipo de modelo de dados a ser utilizado.

As *tags Problem* e *SolutionSpace* definem as instâncias do domínio sobre as quais o *matching* será executado. Para o caso em questão, elas correspondem, respectivamente, à definição de quais pessoas e quais cursos serão processados durante a execução do *matching*.

A *tag Strategies* define as estratégias de *matching* que devem ser utilizadas. Na requisição acima somente a estratégia *BruteForceStrategy* foi escolhida. Para ela foram definidos dois parâmetros: seu nome e a natureza do problema, que nesse caso é de maximização. A *tag DomainKnowledge* define sobre qual domínio o *matching* será executado.

O resultado da execução do problema de *matching* também é definido via um documento XML. Abaixo, na Figura 27, está definido um exemplo de tal documento, que corresponde ao resultado da execução da requisição anterior.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Matchings>
    <Matching model=" ONTOLOGY " name="example" type="1to1">
        <SolutionList name="brute_force">
            <SolutionElement>
                <Problem>
                    <Elements>
                        <Element
id="http://www.lac.inf.puc-rio.br/~ferrao/ontologies/rdf/people/luis.rdf"/>
                    </Elements>
                </Problem>
                <BestSolution>
                    <Solution>
                        <ObjValues><ObjValue order="0" value="1.0"/></ObjValues>
                        <Elements>
                            <Element
id="http://www.lac.inf.puc-rio.br/~ferrao/ontologies/rdf/courses/course2.rdf"/>
                        </Elements>
                    </Solution>
                </BestSolution>
                <AllSolutions>
                    <Solution order="0">
                        <ObjValues><ObjValue order="0" value="1.0"/></ObjValues>
                        <Elements>
```

PUC-Rio - Certificação Digital Nº 0310811/CA

```
                            <Element
id="http://www.lac.inf.puc-rio.br/~ferrao/ontologies/rdf/courses/course2.rdf"/>
                        </Elements>
                    </Solution>
                    <Solution order="1">
                        <ObjValues><ObjValue order="0" value="1.0"/></ObjValues>
                        <Elements>
                            <Element
id="http://www.lac.inf.puc-rio.br/~ferrao/ontologies/rdf/courses/course3.rdf"/>
                        </Elements>
                    </Solution>
                </AllSolutions>
            </SolutionElement>
        </SolutionList>
    </Matching>
</Matchings>
```

Figura 27 – Exemplo de resultado de uma requisição de matching

A *tag Matching* reproduz as informações que foram definidas na requisição do *matching*. Para cada estratégia de *matching*, que foi escolhida na requisição, existe uma *tag SolutionList* correspondente, que descreve a lista de soluções obtida pela execução da referida estratégia.

A *tag Problem* define as entidades do domínio sobre o qual o referido elemento da lista de soluções está relacionado. Também é definida a melhor solução encontrada (*tag BestSolution*), bem como a lista de todas as soluções obtidas (*tag AllSolutions*).

Para cada solução, são definidas as entidades do domínio que a compõem, bem como os valores associados à similaridade com relação à solução em questão (*tag ObjValue*). Tanto as soluções como os valores de similaridade apresentam informação de ordem, sendo possível definir os que possuem maiores níveis de relevância.

4.3. OntoAPI

4.3.1. Diagrama de Classes

A OntoAPI apresenta um amplo conjunto de classes para tratar o acesso a ontologias. Vários *design patterns* foram utilizados em seu projeto, visando a

PUC-Rio - Certificação Digital Nº 0310811/CA

garantir um alto grau de flexibilidade e de extensibilidade. A Figura 28 apresenta uma visão geral das principais classes que compõem a API.

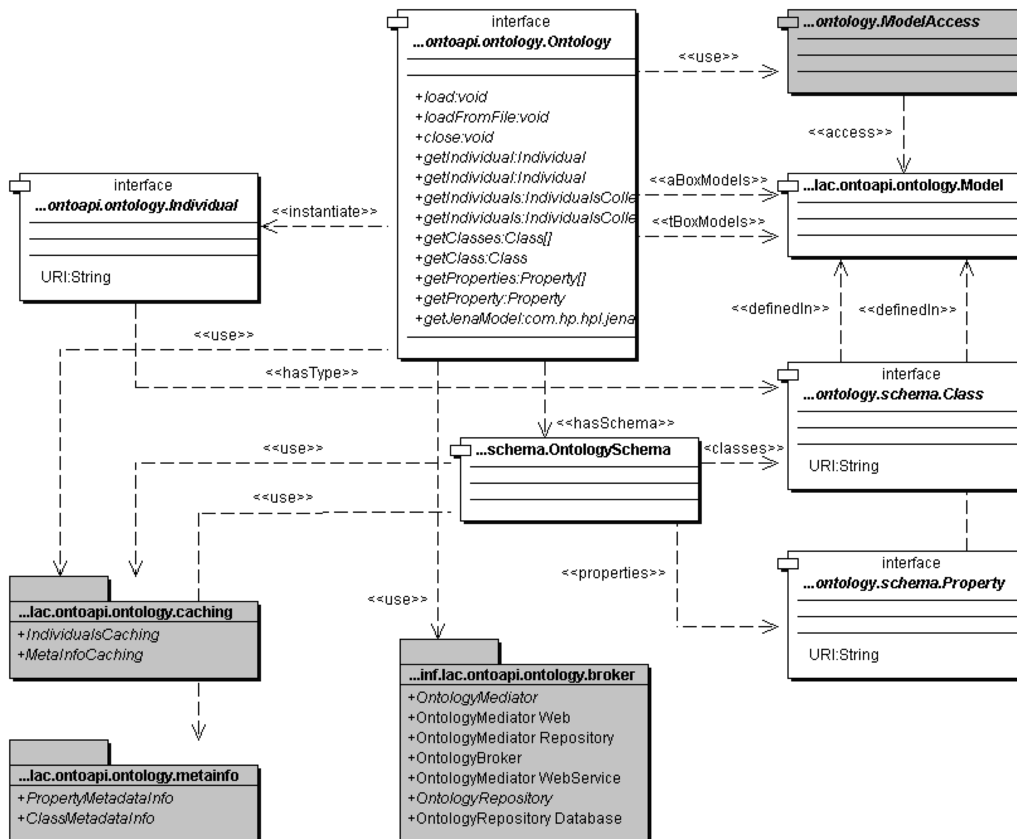

Figura 28 – Diagrama de classes da OntoAPI

A interface *Ontology* representa o ponto principal de acesso às funcionalidades da OntoAPI. Ela define métodos para obtenção das informações relacionadas às instâncias de classes das ontologias, representadas por objetos da interface *Individual*, bem como de suas classes e propriedades, representados por objetos das interfaces *Class* e *Property*.

A interface *Individual* provê à API funcionalidades relacionadas à obtenção dos valores das propriedades das instâncias das classes das ontologias. São definidos métodos para se obter os valores de propriedades cuja imagem sejam tipos primitivos (*strings*, inteiros, dentre outros), bem como para valores de propriedades que representam associações com outras instâncias das classes das ontologias. Para cada uma dessas propriedades são definidos métodos que fazem a conversão apropriada para o tipo de dado em questão.

PUC-Rio - Certificação Digital Nº 0310811/CA

Cada ontologia manipulada pela OntoAPI é representada internamente como um objeto da classe *Model*, cuja linguagem (RDF, RDFS ou OWL) é representada pela classe *Language*. Para fins de organização e de melhoria na performance das consultas, a API separa as ontologias em duas categorias: *tBoxModels* (*Conceptual or Terminological Knowledge*) e *aBoxModels* (*Assertional Knowledge*) [Nardi, 2002]. A primeira categoria inclui ontologias descritas em RDFS e OWL e a segunda categoria corresponde a todas as ontologias que apenas representam instâncias de conceitos, e que estão descritas exclusivamente em RDF.

As informações sobre as classes e as propriedades dos conceitos das ontologias são armazenadas e organizadas na classe *OntologySchema*. Nela estão armazenadas coleções de objetos dos tipos *Class* e *Property*. Além disso, ela usa um conjunto de objetos que visam a obter informações sobre os metadados das ontologias, que são instâncias das classes *ClassMetadataInfo* e *PropertyMetadataInfo*, definidas no pacote *MetaInfo*.

PUC-Rio - Certificação Digital Nº 0310811/CA

O acesso às informações das ontologias é realizado por implementações da interface *ModelAccess*. Com isso, a OntoAPI flexibiliza o processo de escolha dos modelos relevantes para execução de suas consultas, permitindo definições de estratégias mais aderentes às características de um determinado contexto, garantindo, assim, melhor desempenho nas operações de consultas realizadas sobre o mesmo.

Todas as informações obtidas pela OntoAPI, sejam elas sobre as instâncias dos conceitos ou sobre seus metadados, são armazenadas em *cache*, visando a otimizar o processo de consulta. A API disponibiliza duas interfaces, *IndividualsCaching* e *MetaInfoCaching*, que podem ser utilizadas para definições de diferentes estratégias de *caching* para as consultas realizadas sobre os dados das ontologias. Essas interfaces estão definidas no pacote *Caching*.

A obtenção das ontologias é feita sobre demanda e de forma transparente para o usuário da OntoAPI. As classes responsáveis por esse processo estão definidas no pacote *Broker*. Nele está definida a classe *OntologyBroker*, que disponibiliza métodos para localização das ontologias, bem como a classe abstrata *OntologyMediator*, que define um ponto de implementação para instanciação de mediadores utilizados para localizar ontologias nos mais

diversificados ambientes. Esse pacote inclui, ainda, uma interface *OntologyRepository*, que define um conjunto de operações relacionado ao armazenamento das ontologias obtidas através do *broker*.

4.3.2. Executando consultas com a OntoAPI

Esta seção apresenta uma visão geral do processo de consulta às instâncias das classes da ontologia, bem como de suas propriedades. A Figura 29 apresenta um diagrama de classes mais detalhado relacionado às classes da API que são responsáveis por obter tais tipos de informação.

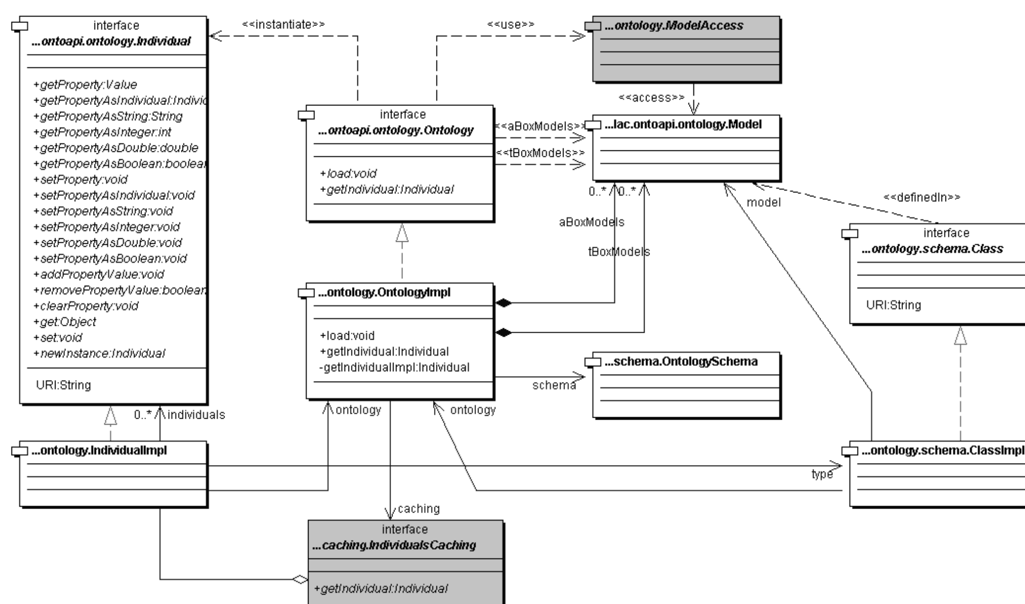

Figura 29 – Diagrama de classes relacionado à consulta por entidades do domínio

A obtenção de uma instância de uma classe da ontologia é definida pela chamada ao método *getIndividual* da interface *Ontology*. O diagrama de seqüência da Figura 30 apresenta o fluxo de operações entre os componentes da API, que correspondem à execução da consulta.

PUC-Rio - Certificação Digital Nº 0310811/CA

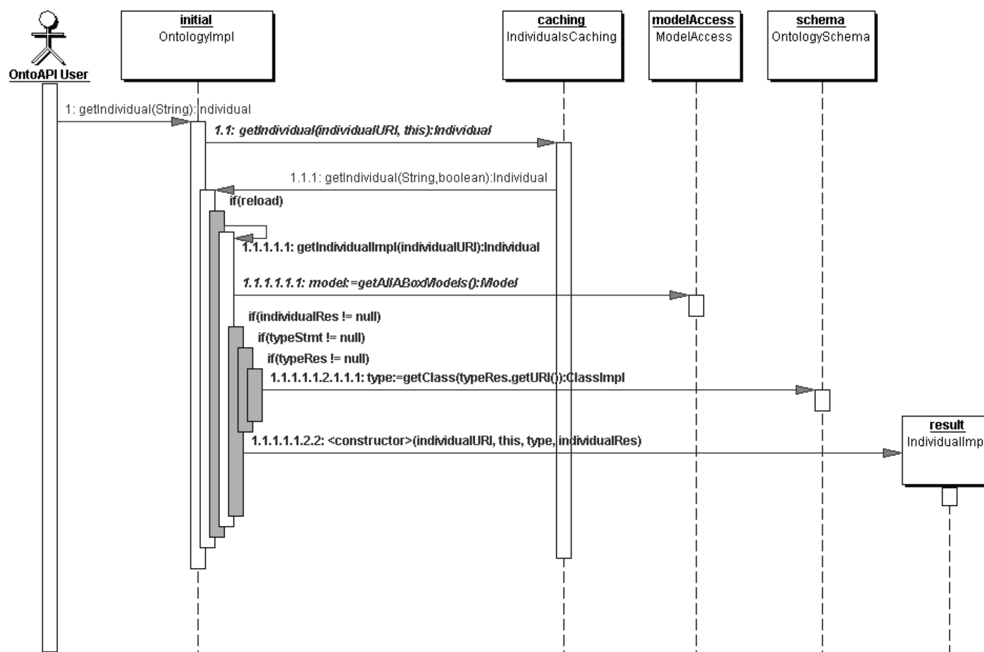

Figura 30 – Diagrama de seqüência relacionado à consulta por entidades do domínio

PUC-Rio - Certificação Digital Nº 0310811/CA

Como pode ser visto, o primeiro passo do processo é tentar resolver a consulta em nível de *cache*. Caso o indivíduo já tenha sido obtido anteriormente, a instância que está armazenada será retornada. Se a ontologia referente à instância não constar naquelas até então manipuladas pela API, uma chamada ao *broker* será realizada, visando a obter tal ontologia. Uma vez de posse da ontologia, a OntoAPI executa operações de busca sobre o conjunto de ontologias retornado pelo *ModelAccess* para obter a instância procurada, retornando ao seu usuário um objeto do tipo *IndividualImpl*, que define uma implementação para o tipo *Individual*.

De posse do objeto correspondente a uma instância de um dado conceito, o usuário da OntoAPI pode utilizar os métodos definidos na interface *Individual* para consultar os valores das propriedades dessa instância. A API oferece métodos que permitem obter tais valores de forma genérica, utilizando estruturas de dados definidas pela API, bem como métodos que fazem conversões desses valores para tipos apropriados. Mais adiante serão apresentados exemplos de uso da OntoAPI, que visam, dentre outras coisas, a demonstrar o uso de tais métodos.

Antes da execução da busca por uma determinada instância de um conceito ou de um valor associado a uma de suas propriedades, a OntoAPI

aplica sobre o conjunto de ontologias relevantes para a consulta, obtido através da classe *ModelAccess*, um *reasoner* adequado à linguagem sobre a qual as ontologias estão descritas. Dessa forma, a consulta sendo realizada não é executada apenas sobre as instâncias explicitadas na ontologia, mas também sobre as instâncias inferidas.

Uma vez obtido uma instância ou um valor de alguma de suas propriedades, essa informação fica armazenada em *cache*. A OntoAPI disponibiliza duas implementações da interface *IndividualsCaching*, que são as classes *IndividualsCaching_NoCaching* e *IndividualsCaching_Memory*. Elas definem estratégias correspondentes ao não uso do *caching* e ao uso do *caching* em memória.

4.3.3. Obtendo Informações Sobre os Metadados das Ontologias

Todas as classes e propriedades de uma ontologia são armazenadas numa estrutura de dados da OntoAPI, definida pela classe *OntologySchema*. Através dela é possível acessar informações sobre classes ou propriedades, via chamadas aos métodos *getClass* ou *getProperty*, como pode ser visto em seu conjunto de operações.

A classe *OntologySchema* também define um conjunto de operações sobre classes e propriedades. Tais operações permitem obter informações tais como as superclasses de uma classe, o domínio de uma propriedade, ou informações não relacionadas a metadados, tais como qual o número de instâncias de uma determinada classe (essa operação fica restrita apenas a uma consulta sobre as ontologias manipuladas pela OntoAPI).

A Figura 31 apresenta uma visão mais detalhada das classes que compõem a funcionalidade de consulta a metadados.

PUC-Rio - Certificação Digital Nº 0310811/CA

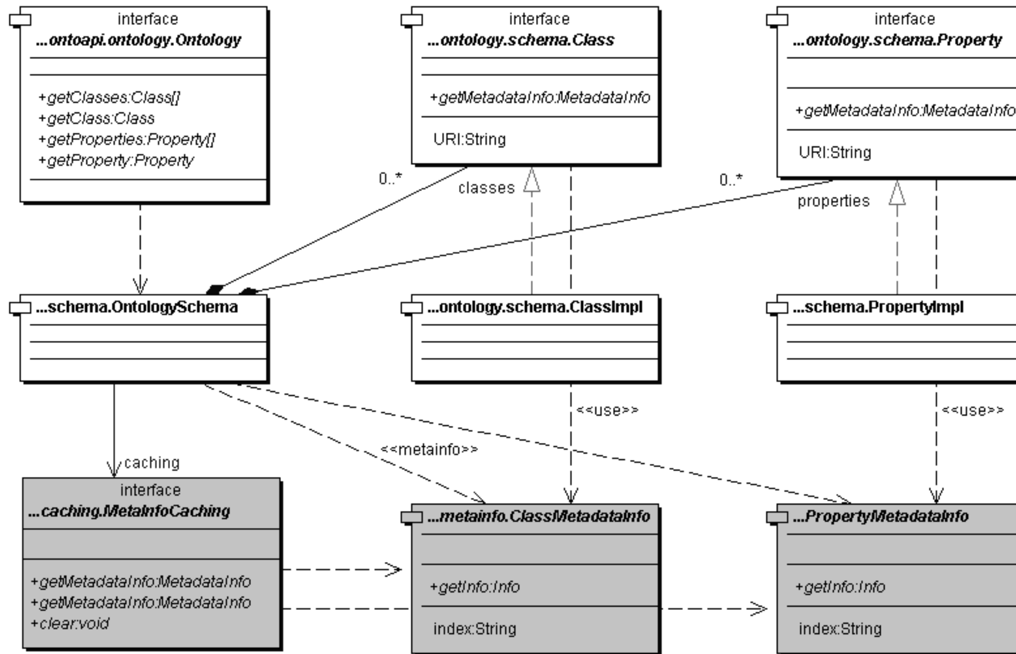

Figura 31 – Diagrama de classes relacionado à consulta por metadados

PUC-Rio - Certificação Digital Nº 0310811/CA

De forma similar ao que acontece com o caso das consultas às instâncias das ontologias, a OntoAPI também disponibiliza um mecanismo de *caching* para os resultados de consultas aos metadados das ontologias. A interface *MetaInfoCaching* e suas implementações *MetaInfoCaching_NoCaching* e *MetaInfoCaching_Memory* são responsáveis pela definição de tal funcionalidade.

A OntoAPI disponibiliza algumas implementações das classes *ClassMetadataInfo* e *PropertyMetadataInfo*. Essas implementações cobrem os casos mais comuns, que tratam, principalmente, de hierarquia de classes e de propriedades, bem como de cardinalidade, domínio e imagem de propriedades.

4.3.4.
Ontology Broker

Uma das funcionalidades mais interessantes da OntoAPI é a localização de ontologias. Ela é executada sempre que uma instância ou uma propriedade referencia uma ontologia que ainda não esteja sendo manipulada pela API. A execução dessa localização é realizada pela classe *OntologyBroker*, com o auxílio de um conjunto de mediadores, que são implementações da classe abstrata *OntologyMediator*.

A Figura 32 apresenta o diagrama de classes dos componentes de *software* da OntoAPI que compõem a funcionalidade de *brokering* de ontologias.

Figura 32 – Diagrama de classes relacionado ao broker de ontologias

PUC-Rio - Certificação Digital Nº 0310811/CA

As classes que processam consultas na OntoAPI usam o método *loadModel* de *OntologyBroker* para solicitar a requisição de localização de uma ontologia. Essa requisição é, então, repassada aos mediadores associados ao *broker*, para que esses tentem localizar a ontologia solicitada no ambiente particular de responsabilidade de cada um deles.

O relacionamento entre o *OntologyBroker* e as várias implementações de *OntologyMediator* é definido segundo o *design pattern Chain of Responsibility*. Dessa forma, caso um mediador falhe no processo de localização da ontologia ele repassa o pedido ao seu sucessor. Esse processo se repete até que algum mediador encontre a ontologia ou até que o último mediador seja executado. A ordenação da cadeia de mediadores é definida via configuração, como poderá ser visto no decorrer desse capítulo.

A classe abstrata *OntologyRepository* define um conjunto de métodos que podem ser implementados visando a garantir o armazenamento das ontologias obtidas via *broker*. Esse armazenamento local possibilita que requisições futuras

às ontologias possam ser resolvidas localmente, sem necessidade de consultas a fontes externas. A OntoAPI disponibiliza uma implementação dessa classe para armazenamento de ontologias em banco de dados, representada pela classe *OntologyRepository_Database*.

Uma questão de projeto bastante interessante é que o acesso aos dados contidos no repositório não é feito diretamente pelo *OntologyBroker*, mas através de um *Mediator* próprio para o repositório. Com essa decisão, obteve-se maior uniformidade no processo de localização das ontologias, deixando-o sempre a cargo do conjunto de mediadores.

A Figura 33 apresenta o diagrama de seqüência relacionado ao processo de *brokering* de ontologias.

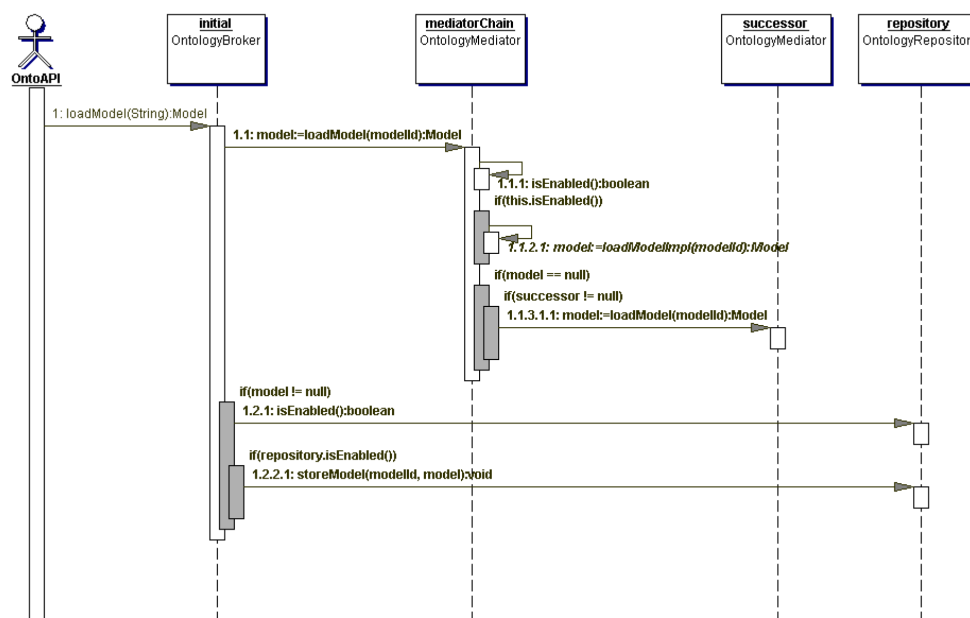

Figura 33 – Diagrama de seqüência relacionado ao processo de brokering de ontologias

4.3.5. Processando uma Nova Ontologia

Para que uma ontologia, que tenha sido requisitada ao *broker*, possa começar a ser utilizada pela OntoAPI, faz-se necessário executar uma série de processos internos, que visam a criar as estruturas de dados que serão utilizadas posteriormente para execução das operações sobre essa ontologia.

PUC-Rio - Certificação Digital Nº 0310811/CA

A primeira dessas estruturas corresponde à criação de um objeto da classe *Model*, que encapsula a ontologia. Além de disponibilizar métodos para acesso às informações constantes numa ontologia, essa classe mantém algumas relações com outros objetos desse mesmo tipo, como discutido abaixo:

- *Relação instances*: Associa um *Model* (ontologia) a outros objetos *Model*, que sejam instâncias do primeiro;
- *Relação extensions*: Associa um *Model* a outros objetos *Model*, que estendam alguma classe ou propriedade definida no primeiro;
- *Relação imports*: Associa um *Model* a outros objetos *Model*, que estejam definidos na propriedade *imports* da ontologia encapsulada pelo primeiro;

Existem também relações inversas para cada uma das relações citadas anteriormente. Essas relações são importantes na medida em que podem ser utilizadas pelas implementações da interface *ModelAccess* para obtenção de subconjuntos relevantes de ontologias para execução de uma dada operação a ser executada pela OntoAPI.

Visando a geração desses objetos *Model*, bem como de suas relações, a OntoAPI define um conjunto de componentes que tem como finalidade a execução de tal processo. O diagrama de classes da Figura 34 apresenta uma visão geral de tais componentes.

PUC-Rio - Certificação Digital Nº 0310811/CA

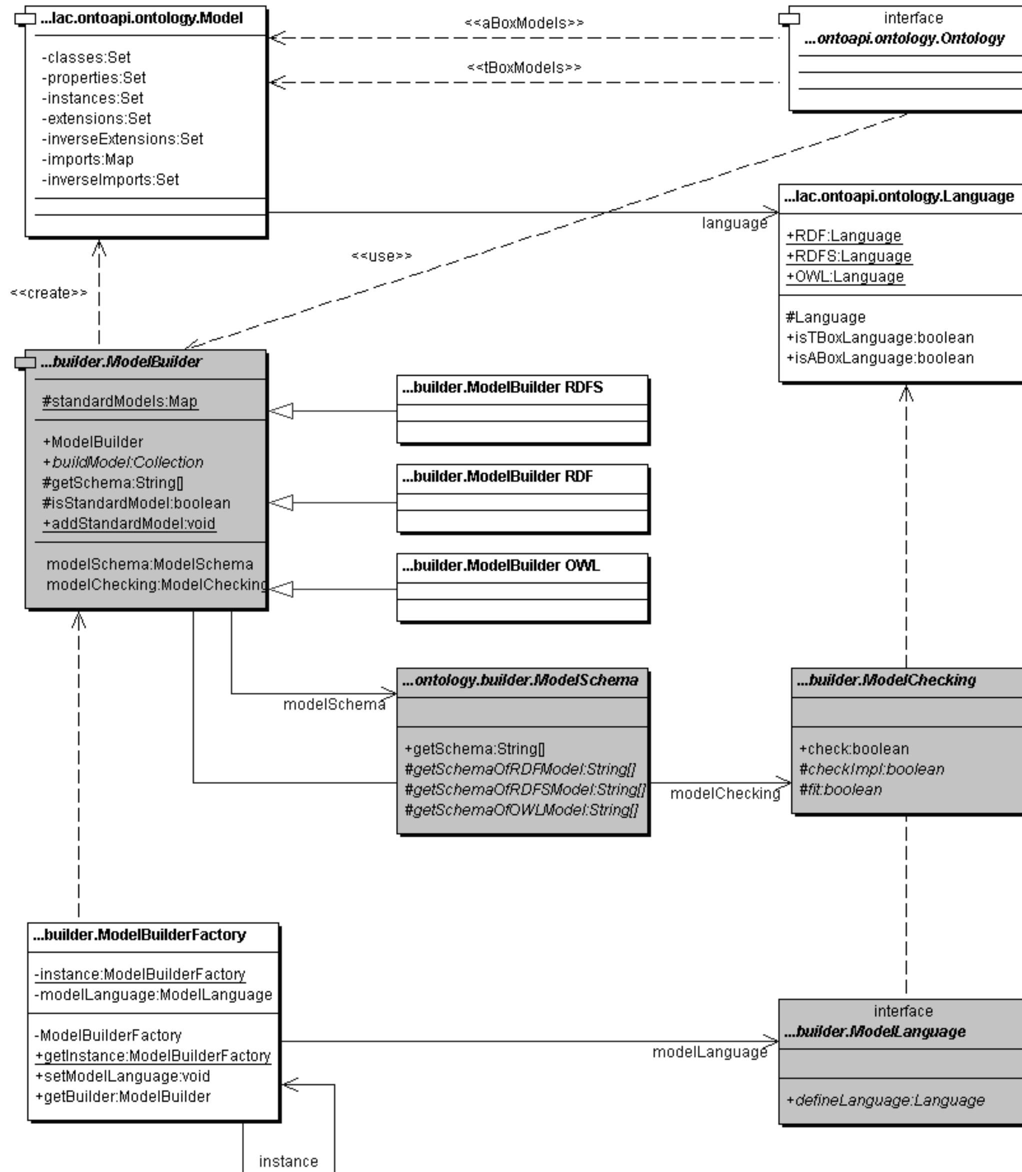

Figura 34 – Diagrama de classes relacionado ao processo de incorporação de ontologias

A classe abstrata *ModelBuilder* é responsável pelo processo de criação de objetos da classe *Model*. Existem implementações de *builder* para cada uma das linguagens suportadas pela OntoAPI: *ModelBuilder_RDF*, *ModelBuilder_RDFS* e *ModelBuilder_OWL*.

A escolha de qual das implementações de *ModelBuilder* utilizar é realizada pela classe *ModelBuilderFactory*. Ela recebe a ontologia solicitada ao *broker* e usa uma implementação da interface *ModelLanguage* para determinar sobre qual linguagem a ontologia que será processada foi descrita. Um objeto *ModelLanguage* define uma estratégia de análise de uma dada ontologia, buscando definir sobre qual linguagem ela está descrita.

PUC-Rio - Certificação Digital Nº 0310811/CA

Durante o processo de geração de uma representação de um *Model*, faz-se uso de outros dois componentes que definem com quais outras ontologias a ontologia sendo processada está relacionada, bem como métodos a partir dos quais se pode verificar se a ontologia pode ser processada pela API. Esses componentes são representados, respectivamente, por implementações das classes abstratas *ModelSchema* e *ModelChecking*.

Quando uma ontologia sendo processada referencia uma outra ontologia, essa última também deve ser manipulada pela OntoAPI. Caso não seja representada por nenhum *Model*, ela será requisitada ao *broker* e incorporada à OntoAPI. Perceba, então, que o processo de criação de um *Model* é recursivo, terminando apenas quando todas as ontologias necessárias forem obtidas e representadas adequadamente.

Um segundo processo relacionado à incorporação de ontologias que definem classes e propriedades é a atualização do conjunto de classes e de propriedades mantido pela classe *OntologySchema*. A OntoAPI define uma classe abstrata *UpdateSchema*, da qual podem ser derivadas implementações que definem estratégias para geração adequada de representações de objetos *Class* e *Property* para representar, respectivamente, as classes e as propriedades definidas na ontologia sendo incorporada.

PUC-Rio - Certificação Digital Nº 0310811/CA

4.3.6.
Configurando a OntoAPI

A OntoAPI disponibiliza um arquivo de configuração, no qual é possível definir todos os parâmetros relacionados aos seus processos. Esses parâmetros dizem respeito, dentre outras coisas, às classes que devem ser utilizadas como implementações dos pontos de flexibilização da API, à cadeia de mediadores que será utilizada para localizar ontologias, bem como sobre os parâmetros que devem ser utilizados pelo repositório associado ao *broker*.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ontoapi-config>

    <ontology
        update-schema="full_package_name.UpdateSchema_Default"
        metainfo-caching=" full_package_name.MetaInfoCaching_Memory"
        individuals-caching=" full_package_name.IndividualsCaching_Memory"
        model-access=" full_package_name.ModelAccess_Default"
        property-checking=" full_package_name.PropertyChecking_Default">
        ...
    </ontology>

</ontoapi-config>
```

Figura 35 – Parâmetros do arquivo de configuração da OntoAPI

O trecho do arquivo de configuração da Figura 35 define os parâmetros relacionados às implementações das estratégias de *cachings* (*tags individuals-caching* e *metainfo-caching*), do algoritmo de atualização dos metadados das ontologias (*tag update-schema*), do algoritmo de escolha das ontologias relevantes para uma consulta (*tag model-access*), dentre outros.

PUC-Rio - Certificação Digital Nº 0310811/CA

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ontoapi-config>

    <ontology ...>

        <model-builder
             schema="full_package_name.ModelSchema_Default"
             checking="full_package_name.ModelChecking_Default">

             <factory language="full_package_name.ModelLanguage_Default"/>

             <standard-model id="http://www.w3.org/2000/01/rdf-schema"/>
             <standard-model id="http://www.w3.org/1999/02/22-rdf-syntax-ns"/>
             <standard-model id="http://purl.org/dc/elements/1.1/"/>
             <standard-model id="http://purl.org/rss/1.0/"/>
             <standard-model id="http://www.daml.org/2001/03/daml+oil"/>
             <standard-model id="http://www.w3.org/2001/vcard-rdf/3.0"/>
             <standard-model id="http://www.w3.org/2002/07/owl"/>
             <standard-model id="http://jena.hpl.hp.com/2003/08/jms"/>
             <standard-model id="http://www.w3.org/2000/10/XMLSchema"/>
             <standard-model id="http://www.w3.org/2001/XMLSchema"/>

        </model-builder>
        ...
    </ontology>

</ontoapi-config>
```

Figura 36 – Parâmetros de configuração relacionados ao construtor de modelos

A *tag model-builder*, apresentada no arquivo de configuração da Figura 36, define os parâmetros relacionados ao processo de incorporação de novas ontologias para serem manipuladas pela OntoAPI. Ela define quais as implementações dos algoritmos relacionados a esse processo devem ser

utilizados pela instância da API, bem como quais ontologias devem ser consideradas como modelos padrão (*standard-models*).

Um modelo padrão se refere a uma ontologia que não precisa ser localizada e nem ter representação como um objeto da classe *Model*. O *framework* Jena possui representações internas de tais ontologias, uma vez que muitas delas nem são acessíveis via seus endereços *Web*. A opção por tornar a definição dessas ontologias como um parâmetro da OntoAPI foi uma questão de flexibilizar a possível definição de outras ontologias como *Standard Model*, não restringindo esse *status* apenas àquelas definidas pelo Jena.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ontoapi-config>

    <ontology ...>
        ...
        <broker>
            <repository enabled="false"
                class="full_package_name.OntologyRepository_Database">
                caching="true"

                <param name="DB" value="MySQL"/>
                <param name="URL" value="jdbc:mysql://localhost/ontoapi-db"/>
                <param name="DRIVER" value="com.mysql.jdbc.Driver"/>
                <param name="USER" value="ontoapi_user"/>
                <param name="PASSWORD" value="ontoapi_password"/>
            </repository>

            <mediators>
                <mediator enabled="true"
                    class="full_package_name.OntologyMediator_Repository"
                    order="0">
                </mediator>

                <mediator enabled="true"
                    class="full_package_name.OntologyMediator_Web"
                    order="1">
                </mediator>
            </mediators>

        </broker>
        ...
    </ontology>

</ontoapi-config>
```

Figura 37 – Parâmetros de configuração relacionados ao broker de ontologias

O trecho do arquivo de configuração da Figura 37 apresenta os parâmetros de configuração do *broker* de ontologias. Ele define a implementação do repositório que deve ser utilizada (*tag repository*), seus parâmetros, bem como se ela deve estar ativa durante a execução da instância da OntoAPI. A *tag mediators* define os mediadores que devem ser utilizados. Para cada mediador é definida a classe que o implementa, a sua ordem na execução da localização da ontologia, bem como se ele está ou não habilitado.

PUC-Rio - Certificação Digital Nº 0310811/CA

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ontoapi-config>

    <ontology ...>
        ...
        <class-metainfo
            index="DIRECT_PROPERTIES"
            class="full_package_name.ClassMI_DirectProperties"/>

        <class-metainfo
            index="DIRECT_SUB_PROPERTIES"
            class="full_package_name.ClassMI_DirectSubClasses"/>
        ...

        <property-metainfo
            index="DIRECT_SUB_PROPERTIES"
            class="full_package_name.PropertyMI_DirectSubProperties"/>

        <property-metainfo
            index="RANGE"
            class="full_package_name.PropertyMI_Range"/>

        <property-metainfo
            index="CARDINALITY"
            class="full_package_name.PropertyMI_Cardinality"/>
        ...

    </ontology>

</ontoapi-config>
```
Figura 38 – Parâmetros de configuração relacionados ao acesso aos metadados

PUC-Rio - Certificação Digital Nº 0310811/CA

Como últimos parâmetros de configuração da OntoAPI, a Figura 38 apresenta o trecho do arquivo de configuração em que são definidas quais informações podem ser obtidas dos metadados das ontologias (*tags class-metainfo* e *property-metainfo*). Para cada uma dessas informações é definido o índice (nome) através da qual a mesma será referenciada, bem como a classe que implementa a consulta aos metadados para obtenção da informação.

A OntoAPI define um componente que é responsável pela leitura, instanciação de classes e definição dos parâmetros da API nos objetos dessas classes. Esse processo acontece sempre que uma nova instância é requisitada por um usuário que deseje utilizar os seus serviços, durante a chamada ao método *createOntology* da classe *OntologyFactory*.

4.3.7.
Exemplo de Uso da OntoAPI

Essa seção apresenta um pequeno exemplo que ilustra alguns aspectos do uso da OntoAPI. O trecho de código da Figura 39 obtém uma instância da API a partir da classe *OntologyFactory* e lê explicitamente um conjunto de ontologias que descrevem pessoas, que são ou empregados ou *trainees*. São,

então, listadas as URI's e os nomes de todas as pessoas, apenas das pessoas que são empregados e, por fim, apenas das pessoas que são *trainees*.

```
public class GetIndividualsExample
{
    public static void main(String[] args)
    {
        try
        {
            OntologyFactory ontFactory = OntologyFactory.getInstance();
            Ontology ontology = ontFactory.createOntology();

            ontology.load(
                "http://www.lac.inf.puc-rio.br/~ferrao/onts/luis.rdf" );
            ontology.load(
                "http://www.lac.inf.puc-rio.br/~ferrao/onts/joao.rdf" );
            ontology.load(
                "http://www.lac.inf.puc-rio.br/~ferrao/onts/antonio.rdf" );
            ontology.load(
                "http://www.lac.inf.puc-rio.br/~ferrao/onts/maria.rdf" );
            ontology.load(
                "http://www.lac.inf.puc-rio.br/~ferrao/onts/tereza.rdf" );

            IndividualsCollection inds = ontology.getIndividuals( "Person" );
            System.out.println( "Person list:" );
            printIndividuals( inds );

            inds = ontology.getIndividuals( "Employee" );
            System.out.println( "Employee list:" );
            printIndividuals( inds );

            inds = ontology.getIndividuals( "Trainee" );
            System.out.println( "Trainee list:" );
            printIndividuals( inds );
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
    }

    public static void printIndividuals( IndividualsCollection individuals )
            throws Exception
    {
        Iterator it = individuals.iterator();
        while( it.hasNext() )
        {
            Individual i = (Individual) it.next();
            System.out.println(
                i.getURI() + ">> " +
                i.getPropertyAsString( " name" ) );
        }
    }
}
```

Figura 39 – Exemplo de uso da OntoAPI

PUC-Rio - Certificação Digital Nº 0310811/CA

Observe que, nesse exemplo:

- As ontologias usadas não necessitam ser, necessariamente, aquelas disponíveis na *Web*, mas podem representar versões armazenadas localmente no repositório de ontologias;
- As instâncias de uma determinada classe são obtidas via um conjunto de objetos da classe *Individual*, representados por um objeto da classe *IndividualsCollection*. Essa classe disponibiliza um método para iterar sobre todos os objetos agregados a ela;
- Durante a listagem de todas as pessoas, a OntoAPI utilizou a relação *subClassOf* para inferir que um empregado e um *trainee* também são pessoas;
- As classes e as propriedades não necessitam ser especificadas por sua URI completa nas chamadas dos métodos da OntoAPI. Podem haver situações em que seja necessário especificar o valor completo da URI, pois duas classes podem ter o mesmo *label*, ocorrendo nesse caso um impasse;
- Existem métodos que fazem a conversão de tipo explícita para o valor apropriado de uma propriedade, como pode ser observado na chamada ao método *getPropertyAsString* da interface *Individual*.

PUC-Rio - Certificação Digital Nº 0310811/CA

4.4. Integração entre o Matching Module e a OntoAPI

A OntoAPI define a solução de *software* utilizada pelo *framework* Matching Module para trabalhar com modelos de dados baseados em ontologias. A integração desses dois componentes é definida via a implementação das classes do *framework* relacionadas ao acesso a modelos de dados, ou seja, às classes *ModelCreator*, *Model* e *Individual*. O diagrama de classes da Figura 40 apresenta uma visão geral dessa integração.

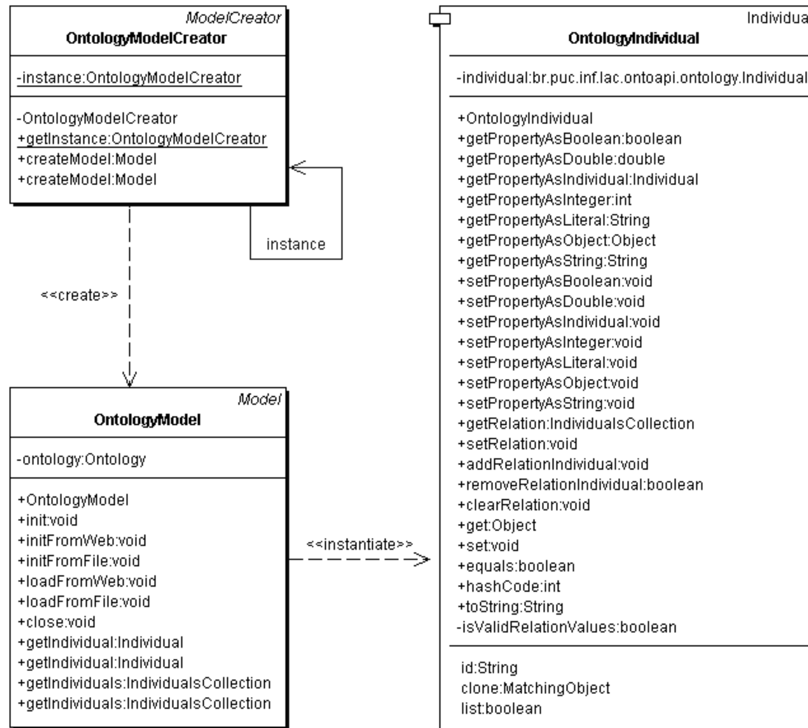

Figura 40 – Classes que compõem a integração entre o Matching Module e a OntoAPI

O projeto das classes segue o *design pattern Adapter*. As implementações são construídas de acordo com as definições do *framework* Matching Module, fazendo chamadas para os métodos apropriados da OntoAPI. Antes de cada uma dessas chamadas, os parâmetros e valores utilizados são convertidos de acordo com aqueles esperados pela API. Os resultados obtidos são, novamente, convertidos para os formatos de dados do *framework*.

Assim, as classes da OntoAPI não precisam se comprometer com as interfaces do *framework*, permitindo com que seu projeto, implementação e utilização sejam totalmente independentes do uso específico das necessidades relacionadas à infra-estrutura proposta.

PUC-Rio - Certificação Digital Nº 0310811/CA