3
MatchMaking – Uma infraestrutura para alinhamento de esquemas

Este capítulo apresenta o *MatchMaking*, uma infraestrutura para alinhamento de esquemas. A seção 3.1 descreve os detalhes da arquitetura conceitual do sistema, discorrendo sobre as decisões do modelo. A seção 3.2 apresenta em detalhe a implementação do *Matchmaking*, descrevendo seus principais módulos e características.

3.1.
Detalhes da arquitetura do Matchmaking

O *Matchmaking* foi criado com o objetivo de prover uma infraestrutura para facilitar a criação dos algoritmos de alinhamento de esquemas, do ponto de vista do criador do algoritmo, e diminuir a complexidade de uma operação de alinhamento do ponto de vista do usuário da ferramenta. Nessa versão da infraestrutura, esquemas descritos em OWL foram utilizados para o alinhamento. Porém, com poucas alterações na estrutura do software é possível utilizá-lo com outras tecnologias.

A ferramenta deve armazenar a proveniência de cada processo de alinhamento ocorrido. Ou seja, deve acomodar um histórico de todos os alinhamentos, contendo os parâmetros de execução, etapas do algoritmo que foram processadas, funções de similaridade utilizadas e representações de dados aplicadas para que o processo fosse concluído. Esses dados sobre proveniência precisam estar disponíveis para o usuário poder comparar os diferentes resultados obtidos.

Por fim, com o intuito de fomentar a comparação de resultados, o *MatchMaking* deve ser capaz de realizar operações de alinhamento em série, automaticamente. Ou seja, o usuário poderá definir esquemas de origem e de destino (e seus *datasets*) e especificar uma série de operações de alinhamento que devem ser executadas. Após isso, a infraestrutura executará cada operação de

alinhamento especificada a série, de forma automática, sem necessidade de interação adicional com o usuário.

A arquitetura da infraestrutura é descrita através do modelo conceitual de seu banco de dados interno (Figura 2).

Um esquema (*Schema)* é definido como uma agregação de elementos: classes, propriedades e instâncias (*class*, *property* e *instance*, respectivamente). Uma propriedade contextualizada (*CProperty*) define um relacionamento entre classes e propriedades, capturando o fato de que uma mesma propriedade pode estar definida para mais de uma classe do esquema.

Em cada esquema, podemos ter vários conjuntos de dados (*Datasets*) que representam instâncias de classes obtidas em um determinado momento. O *Matchmaking* consegue controlar vários desses conjuntos de forma independente. Nas técnicas de alinhamento extensivas ou híbridas, os valores das propriedades de cada uma das instâncias de classe podem alterar o resultado do alinhamento. Portanto, é importante diferenciar que conjunto de dados foi utilizado em um alinhamento.

No caso das ontologias OWL, um arquivo pode possuir elementos de outras ontologias que não são de interesse no que tange o alinhamento de esquemas. Por isso cada esquema possui uma lista de *namespaces* válidos (*validNamespaces*) para aquele esquema no contexto do *MatchMaking*.

Um algoritmo de alinhamento (*Matcher*) possui uma lista de parâmetros que deve, obrigatoriamente, ser preenchida no momento da sua execução (*Execution*). Ele também pode aplicar (*Applies*) funções de similaridade (*SimilarityFunction*) em representações dos dados do esquema (*SetType*).

O algoritmo pode ser dividido em vários *passos* (também chamados de subalgoritmos), cada qual considerado como um novo procedimento com parâmetros e aplicações de funções de similaridade próprias. Com essa arquitetura, é possível até mesmo criar um procedimento que agrega outros algoritmos independentes e calcula a similaridade de dois esquemas através do resultado obtido pela execução dos outros. Por exemplo, com o resultado obtido entre dois algoritmos de alinhamento, podemos aplicar uma equação que calcula o maior, menor ou média dos dois valores e considerar esse o grau de similaridade. É possível diferenciar se um *Matcher* é um algoritmo independente ou não através do atributo *steponly* (que indica para a ferramenta se o procedimento pode ser

executado independentemente ou somente por outro algoritmo).

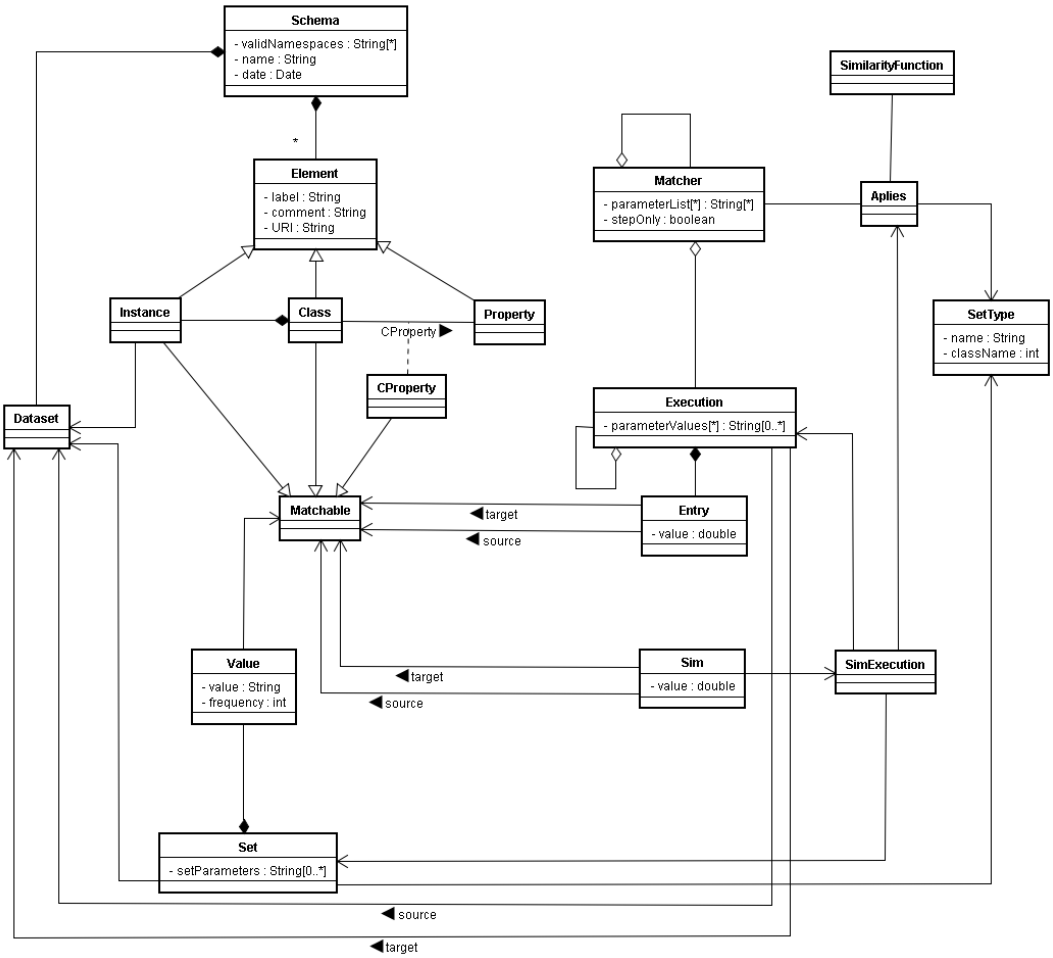

Figura 2 - Modelo conceitual do banco de dados.

Um *elemento alinhável* (*Matchable)* é um elemento do esquema que pode ser comparado com os demais elementos alinháveis do outro esquema. Já uma *execução* é um conjunto de entradas (*Entry*) que representam o conjunto de pares de elementos que foram alinhados ao final da execução. A execução também armazena as informações de que outras execuções ocorreram sob seu contexto (os *passos*) além de oferecer a informação de que funções de similaridade foram aplicadas no decorrer do procedimento.

Uma *representação dos dados* (*SetType*) é uma definição de como os dados de um determinado *dataset* devem ser representados (*Set*). Essa definição pode ser parametrizada de modo que a execução de uma determinada função de similaridade (*SimExecution)* pode solicitar a geração desse conjunto de forma diferente em cada execução. Os dados obtidos (*Value*) estão diretamente

relacionados a um elemento alinhável (*Matchable*).

A execução de uma função de similaridade utiliza os valores gerados em uma representação dos dados para calcular a similaridade entre dois elementos alinháveis (*Sim)*. Todo elemento comparado por uma função de similaridade é registrado no banco de dados.

Podemos, então, dividir a arquitetura do sistema em três diferentes segmentos (Figura 3): esquema, algoritmo e execução.

A primeira parte, *esquema*, é responsável por definir a estrutura de um esquema no banco de dados da ferramenta. Essas informações presentes são suficientes para que os algoritmos consigam entender um esquema. Apesar disso, outras informações sobre o esquema podem ser requeridas pelo algoritmo. Para isso, existe a possibilidade de utilizar uma ferramenta de *parsing* de esquemas OWL (maiores detalhes na seção 3.2).

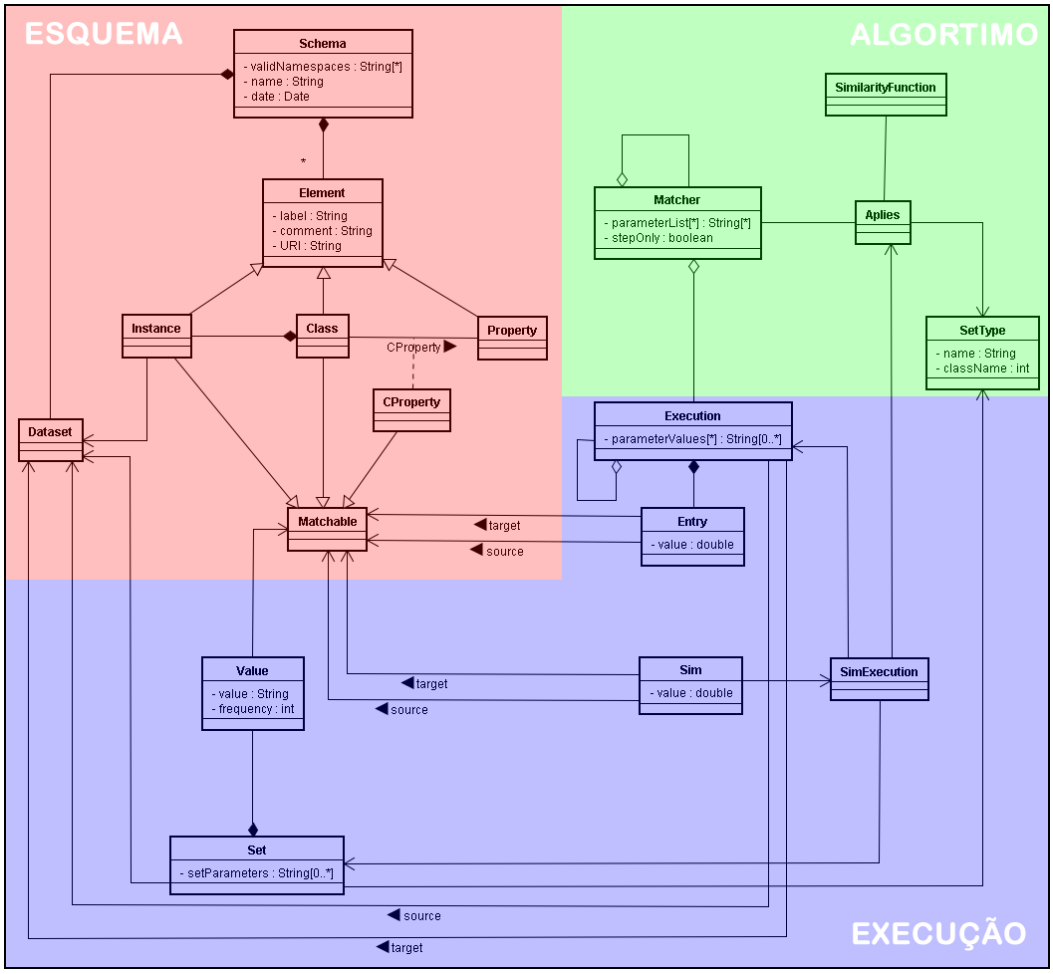

Figura 3 - Divisão da arquitetura.

Já a segunda parte, *algoritmo*, é a responsável por representar um algoritmo dentro do banco de dados. O algoritmo e seus parâmetros são definidos em *Matcher* enquanto seu relacionamento com as funções de similaridade são definidos através do relacionamento ternário *Applies*.

A última parte contem as informações da *execução* de um determinado algoritmo. A execução, os valores de parâmetros, que funções de similaridade foram utilizadas e que resultados foram gerados são registrados nesse setor. Também são informados que *representações de dados* (*Set*) foram lidas. Ou seja, nesse segmento estão as informações de proveniência do sistema.

Essa seção apresentou a arquitetura conceitual do sistema, com seus detalhes e intenções. A próxima seção mostra como essa modelagem foi desenvolvida para funcionar na prática.

3.2. Implementação do Matchmaking

3.2.1. Requisitos

Alguns requisitos foram definidos de modo a servir de guia para o desenvolvimento do *Matchmaking*:

- A infraestrutura não deve ser independente. Ou seja, ela não deve ser capaz de executar suas operações sozinha. Todas elas devem ser executadas por meio de outro aplicativo que tem o *Matchmaking* embutido em seu código (Figura 4). Por exemplo, a interface gráfica criada no capítulo 4 (Testes do Protótipo) é uma aplicação Web que acessa as funções do *Matchmaking* através da sua biblioteca de classes.

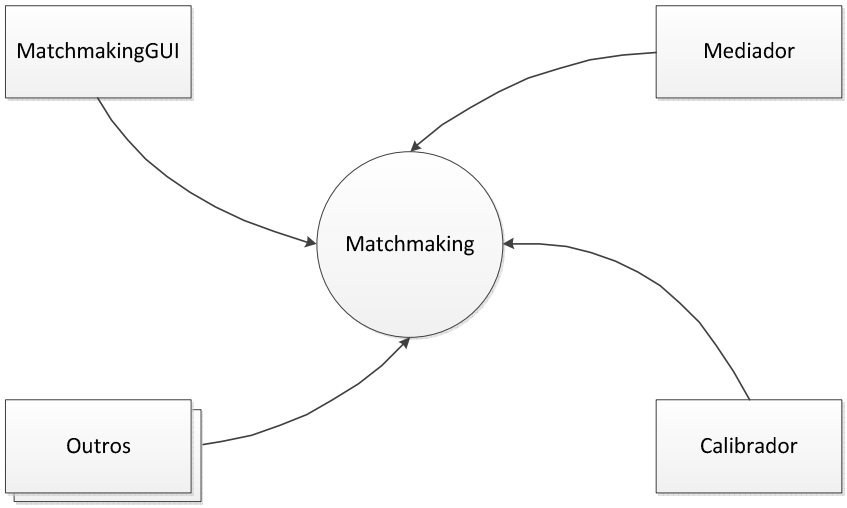

Figura 4 - *Matchmaking* sendo acessado por várias aplicações.

- A infraestrutura deve ser expansível. Ou seja, o *Matchmaking* deve permitir que sejam criados novos algoritmos, funções de similaridade e representações de dados de forma simples e intuitiva.
- A infraestrutura deve permitir acesso completo aos dados dos esquemas que estão sendo alinhados. Existem informações disponíveis em uma ontologia que não são armazenadas no banco de dados da ferramenta (os valores de uma propriedade, por exemplo). Essa informação precisa estar disponível de alguma forma para o algoritmo de alinhamento.

3.2.2. Pacotes principais

Para atender esses requisitos, o sistema foi dividido em sete pacotes principais:

- **matchmaking.beans**: apresentam as classes de *beans*. Ou seja, são classes utilizadas para o transporte de informações entre os pacotes e aplicativos terceiros.
- **matchmaking.commands**: é a superfície do sistema. Aqui se encontram todos os comandos disponíveis para os aplicativos terceiros. Maiores detalhes encontram-se na seção 3.2.3 (Comandos).

- **matchmaking.sets**: contem as classes de representação dos dados. Também está disponível a classe de definição, a qual todas as representações precisam implementar para serem válidas no sistema. Maiores detalhes encontram-se na seção 3.2.4.
- **matchmaking.similarity**: contém as funções de similaridade já implementadas no sistema. Também há a classe de definição que precisa ser estendida para uma função ser reconhecida pela ferramenta. Maiores detalhes encontram-se na seção 3.2.5 (Funções de Similaridade).
- **matchmaking.matcher**: possui os algoritmos de alinhamento já implementados e também a classe de definição. Todos os algoritmos precisam implementar essa classe para serem reconhecidos pela ferramenta. Maiores detalhes na seção 3.2.6 (Algoritmos de Alinhamento).
- **matchmaking.schema**: representa um esquema alinhável. As classes desse pacote são utilizadas pelos algoritmos de alinhamento para obter os dados de um determinado esquema. Maiores detalhes encontram-se na seção 3.2.7 (Esquema).
- **matchmaking.dao**: é a camada de acesso aos dados da ferramenta. Todas as consultas ao banco de dados são realizadas a partir desse pacote. Maiores detalhes encontram-se na seção 3.2.8 (Banco de Dados).

3.2.3. Comandos

O pacote *matchmaking.commands* é a fachada do sistema, permitindo que aplicativos externos acessem as funcionalidades do *Matchmaking*. O Apêndice A apresenta um detalhamento de cada um dos comandos possíveis na ferramenta.

Utilizando o padrão de projetos *Command* [12], cada um dos comandos é isolado em uma classe separada que estende a classe abstrata *AbstractCommand* (Figura 5). Essa classe é responsável por todo o controle transacional e também é capaz de capturar qualquer exceção não prevista, impedindo que o sistema opere de maneira instável.

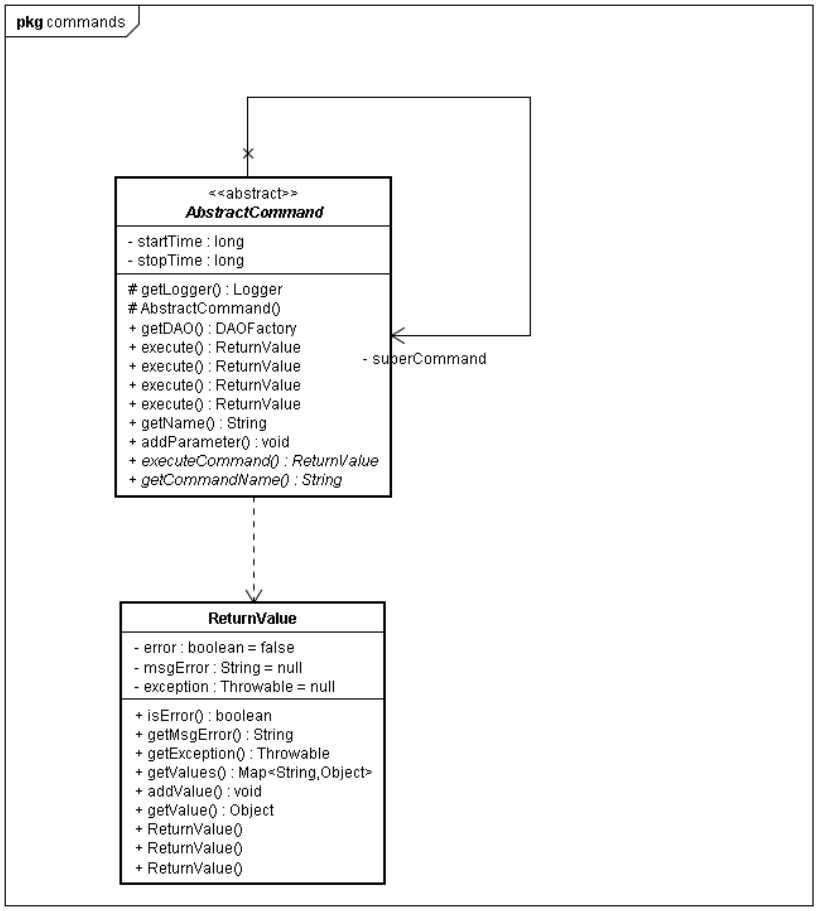

Figura 5 - Implementação dos comandos[1]

Cada execução de um comando segue uma seqüência (Figura 6) inicial e final pré-estabelecida pela classe abstrata *AbstractCommand*: primeiro, o aplicativo externo solicita a execução do comando, criando uma instância da classe desejada, e chama o método *execute* informando uma lista de parâmetros pré-estabelecida para o comando. A *AbstractCommand* então obtém uma conexão com o banco de dados através do método estático *getDaoFactory* da classe *DAOFactory* e executa o método *executeCommand* que precisa ser implementado pela classe de comando concreta.

A classe concreta realiza suas operações e retorna uma instância da classe *ReturnValue* que armazena informações sobre o sucesso ou o insucesso da operação, exceções e valores de retorno.

Por fim, a *AbstractCommand* verifica se houve erro na operação e solicita a confirmação ou cancelamento das operações no banco de dados (*commit* ou *rollback*).

[1] Parâmetros dos métodos foram omitidos.

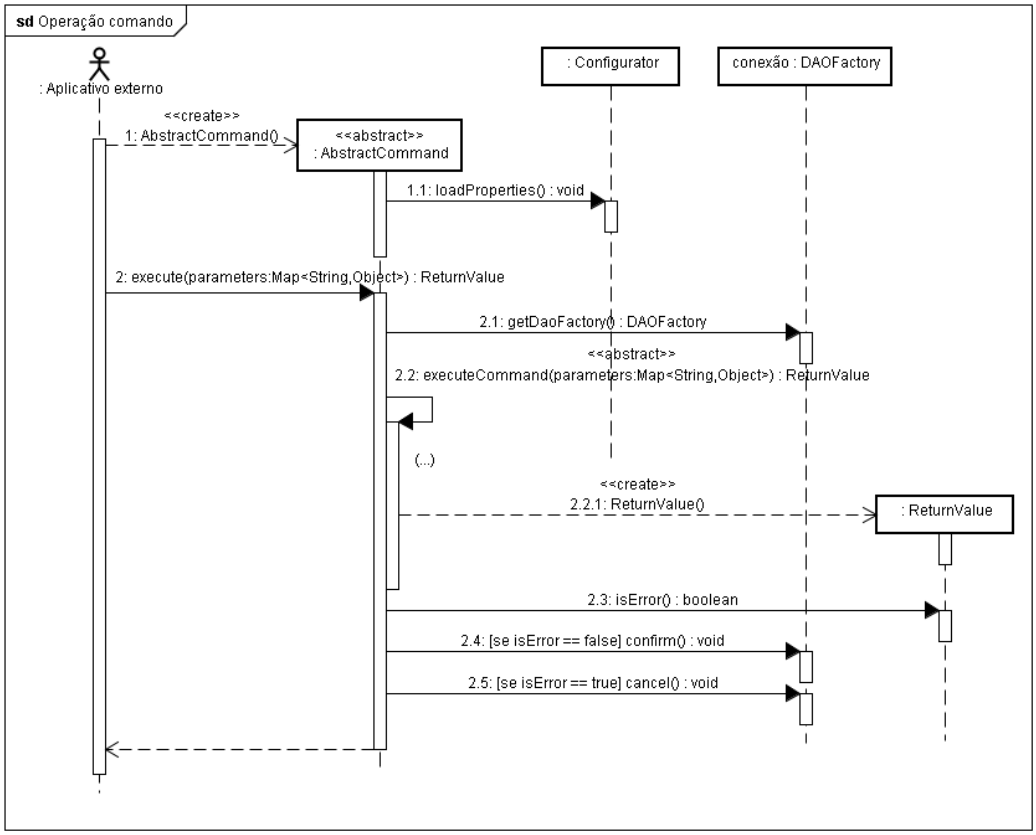

Figura 6 – seqüência genérica de operações de um comando

Para se adequar às necessidades do *Matchmaking*, a implementação também permite a execução de comandos aninhados. Ou seja, um comando que executa outro dentro de uma mesma transação. Por exemplo, o comando *NewSchema* é o responsável por cadastrar um novo esquema no banco de dados e o comando *NewDataSet* é o responsável por salvar um novo *DataSet*. Quando um novo esquema é cadastrado no sistema, obrigatoriamente um novo *DataSet* é também criado. Portanto, o comando *NewSchema* executa o comando *NewDataSet* internamente e, conseqüentemente, se houver algum erro na operação do *NewDataSet*, todo o procedimento será cancelado.

A camada de controle foi divida em três áreas:

- *Execution* – responsável por todos os comandos relativos à execução de um algoritmo de alinhamento e proveniência dos dados.
- *Matcher* – responsável pelo cadastramento e validação de algoritmos de alinhamento.
- *Schema* – responsável pelo cadastramento de novos Schemas, datasets e validações

O Apêndice A. apresenta uma lista com todos os comandos do *Matchmaking*.

3.2.4. Representação de dados

Os dados contidos em um elemento alinhável (*matchable)* de um esquema podem ser representados de diferentes maneiras. A representação trivial do dado seria ele próprio, sem nenhum processamento. Outra maneira seria a criação de um conjunto de *tokens* de modo que, para cada palavra, um novo *token* é criado e a sua freqüência de repetição armazenada. Cada função de similaridade tem a sua necessidade e pode precisar de uma representação dos dados específica.

É importante notar, também, que uma mesma representação de dados pode variar de acordo com alguns parâmetros. Imagine, por exemplo, o caso dos *tokens*. Em alguns casos pode ser interessante descartar palavras com um tamanho muito pequeno, em outros é provável que os valores numéricos devam ser descartados. Tudo depende do domínio dos dados e do que está se buscando.

Dependendo do caso, a geração de uma representação de dados pode ser a operação mais cara (em termos de recursos computacionais) de um processo de alinhamento de esquemas. Afinal, é preciso processar todos os dados contidos em ambos os esquemas analisados para, então, calcular suas similaridades.

Ao observar o funcionamento do processo de geração de uma representação de dados, vemos que o resultado será sempre o mesmo, se repetirmos o esquema e *dataset* analisado e os parâmetros informados. De maneira que essa representação não precisa ser calculada toda vez que for preciso usá-la. Ao ser solicitada a geração de uma nova representação de dados, o *Matchmaking* verifica se esta já não foi processada anteriormente e, caso tenha, apenas informa os valores processados na outra ocasião. Isso permite testes em operações de alinhamento extremamente rápidos e facilita a analise de uma variação de parâmetros muito maior em menos tempo.

Cada função de similaridade pode ser aplicada a uma ou mais representações de dados. Se um desenvolvedor pode criar sua própria função de similaridade no *Matchmaking*, ele também deve poder criar sua própria função de geração de representação de dados.

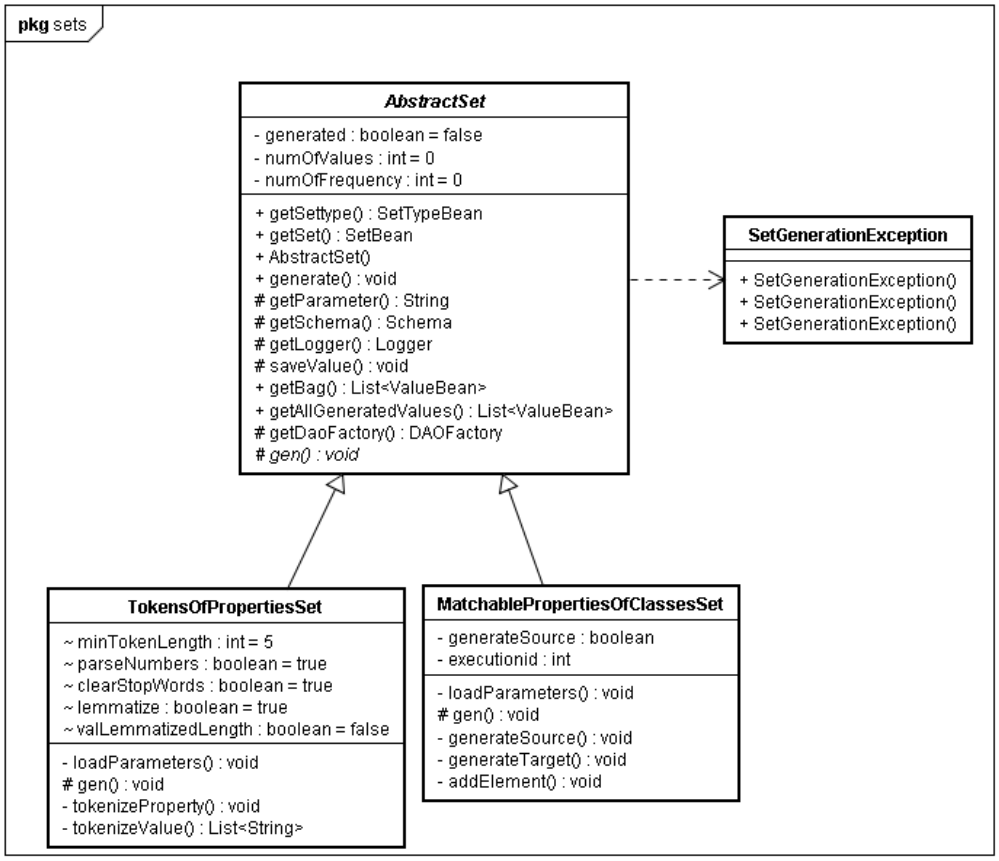

Figura 7 - Representação de dados[2].

A Figura 7 apresenta o diagrama de classes da parte de representação de dados. Nesse caso, existem dois geradores de representação de dados (*TokensOfPropertiesSet* e MatchablePropertiesOfClassesSet) que estendem a classe *AbstractSet* a qual é responsável por fornecer um conjunto de funções para todos os geradores poderem operar, além de obrigá-los a ter uma interface unificada.

A Figura 8 apresenta uma seqüência genérica de operações realizadas quando é solicitada a geração dos dados. A classe *AbstractSet* recebe a solicitação através do método *generate()* que aguarda o esquema a ser calculado, que *dataset* e quais são os parâmetros.

O método verifica no banco de dados se essa representação de dados já não foi calculada. Se não foi, o procedimento chama o método abstrato *gen()* que deve ser implementado pela classe concreta. Esse método deve calcular os valores de acordo com os parâmetros passados e solicitar o salvamento dos dados gerados

[2] Parâmetros dos métodos foram omitidos

(função *saveValue()*). No momento do salvamento dos dados, verifica-se se o valor já existe no banco de dados e, se existir, apenas incrementa a freqüência deste. O método *generate()* retorna da mesma maneira em ambos os casos (se a representação foi gerada nessa execução, ou antes) de modo que o procedimento é completamente transparente.

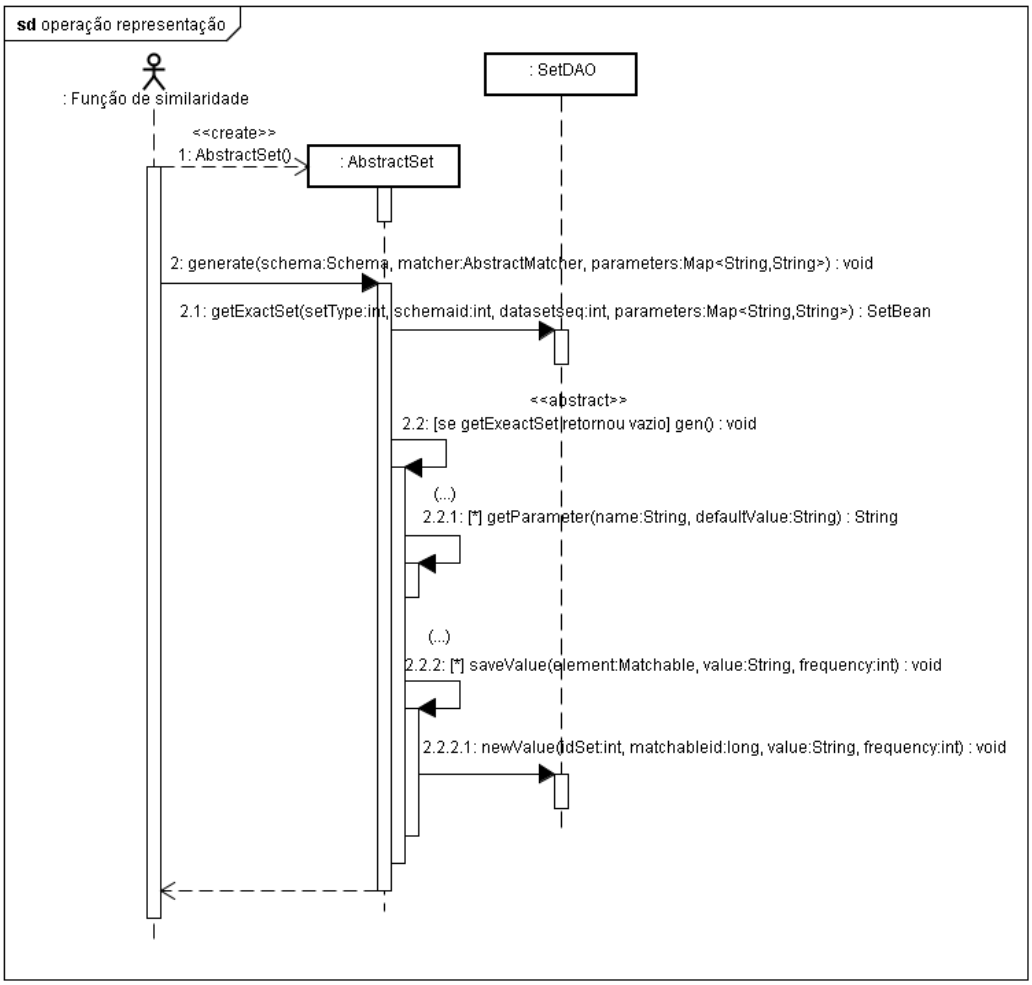

Figura 8 - seqüência genérica da geração de uma representação de dados

3.2.5. Funções de similaridade

Uma função de similaridade é uma função que, dado um elemento de origem e outro de destino, compara o conjunto de valores (através de uma representação de seus dados) de ambos os elementos e atribui um valor que indica o grau de similaridade entre eles.

O suporte a funções de similaridade no *Matchmaking* permite que um desenvolvedor, com objetivo de executar um determinado algoritmo de similaridade, crie suas próprias funções. A maneira de desenvolver uma função de

similaridade é semelhante a uma representação de dados. O programador deve criar uma nova classe e estender a *AbstractSimilarity* (Figura 9) para que sua função seja integrada ao sistema.

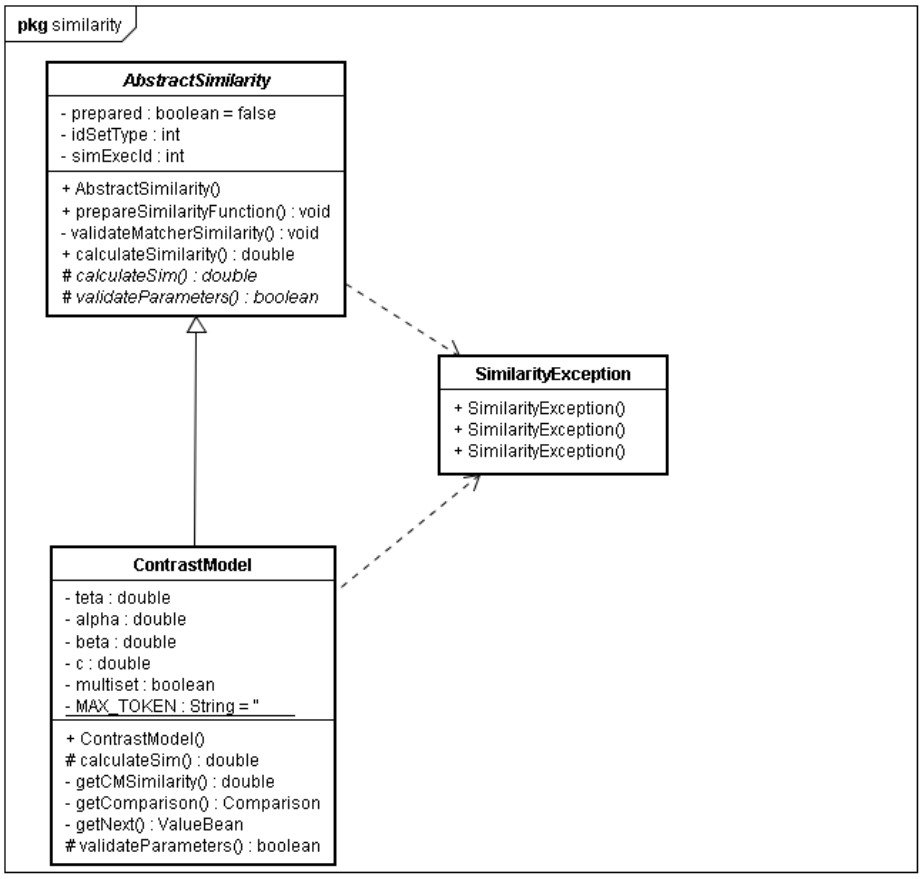

Figura 9 - Funções de similaridade[3].

Já o procedimento envolvido é bastante diferente. O algoritmo não necessariamente precisa calcular a similaridade entre todos os elementos do esquema origem com todos do esquema destino. Dependendo do caso e da otimização, ele pode avaliar apenas alguns casos específicos. Por exemplo, no geral, não há necessidade de se comparar dois elementos que possuam tipos de dados diferentes. Ou talvez seja necessária alguma heurística que indique a compatibilidade entre tipos de dados. Dessa forma, é importante – por questões de proveniência – saber que elementos foram comparados no algoritmo e qual foi o grau de similaridade calculado pela função.

A seqüência de operações apresentada na Figura 10 mostra como um algoritmo deve solicitar o calculo de similaridade. O primeiro passo é ele iniciar

[3] Parâmetros dos métodos foram omitidos

um procedimento de preparação da função (*prepareSimilarityFunction)*. Esse procedimento é feito de forma igual para todas as funções de similaridade e recebe a informação de que execução de algoritmo está solicitando a operação, quais são as representações de dados de origem e destino que serão utilizadas e que valores de parâmetros irão guiar o calculo. É importante notar que essas funções também podem ser parametrizadas. O *Contrast Model*, por exemplo, possui coeficientes (α, β e θ) que podem variar.

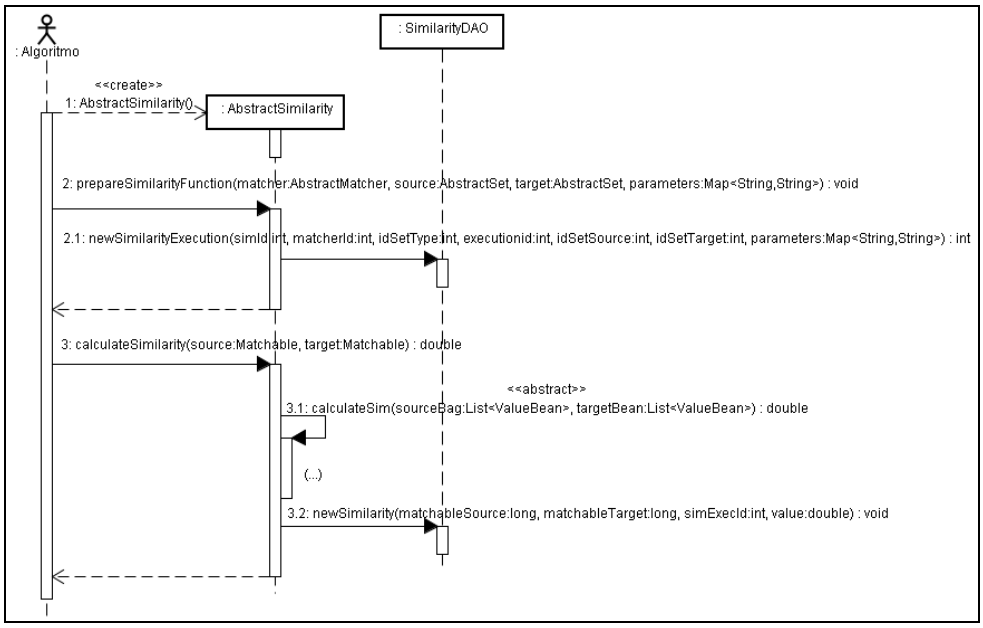

Figura 10 - Seqüência genérica do calculo de uma função de similaridade.

Depois de preparada, a função estará pronta para calcular a similaridade entre dois elementos. Nesse caso, o algoritmo deve solicitar a comparação de elementos um a um. Essa solicitação é realizada através do método *calculateSimilarity*, que repassa para o método abstrato *calculateSim* (implementado de forma diferente para cada função de similaridade). No final, *calculateSimilarity* recebe o grau de similaridade entre os elementos, salva esse dado no banco de dados e retorna o valor para o algoritmo. Essa etapa de gravação dos dados é totalmente transparente e permite ao usuário saber que elementos foram comparados e qual o valor obtido pelo algoritmo utilizando a função de similaridade.

3.2.6. Algoritmos de alinhamento

Algoritmos de alinhamento são a essência do *Matchmaking*. É a execução destes algoritmos e os seus resultados que têm a maior importância na ferramenta. Esses algoritmos devem ser capazes, através da informação dos esquemas e seus dados, de alinhar dois esquemas de bancos de dados.

Ao contrário de funções de similaridade e das representações de dados, o usuário precisa informar, no momento do cadastramento de um algoritmo de alinhamento, quais parâmetros serão precisos para a execução. Além disso, ele também deve informar que funções de similaridade (aplicada a alguma representação de dados) e quais passos o algoritmo utiliza. Apesar dessa obrigatoriedade, a utilização dos passos e das funções de similaridade é opcional. Em alguns casos, um algoritmo pode terminar com o resultado obtido até aquele ponto da execução e desistir de realizar mais alguma operação, por exemplo.

Apenas a declaração dos parâmetros é obrigatória, pois estes são os únicos valores que um usuário deverá especificar ao invocar um algoritmo de alinhamento. Quando o usuário seleciona qual algoritmo executar, os parâmetros devem ser apresentados para serem preenchidos. No caso da função de similaridade e dos geradores de representação de dados, a execução é feita por meio de um algoritmo que foi desenvolvido por um programador o qual possui informações técnicas a respeito dos parâmetros de ambos.

Os passos (ou subalgoritmos) são de fato importantes para o *Matchmaking* pois os algoritmos de alinhamento tipicamente dividem o seu calculo em etapas. O algoritmo proposto por Leme [20] (Capítulo 2.2 e 4) possui uma subdivisão em quatro subalgoritmos principais que geram entradas de alinhamento. Da forma como a ferramenta foi estruturada, cada subdivisão dessa é um passo e possui parâmetros, execuções de funções de similaridade e resultados de alinhamento. Ou seja, para cada subalgoritmo executado no *Matchmaking* há a informação de proveniência. Outro caso da utilização dos passos pode ser a criação de um algoritmo que execute dois ou mais algoritmos diferentes, compara os resultados e combina ambos de forma a criar um alinhamento mais preciso. Existe uma diferença entre os dois casos apresentados: no primeiro, os subalgoritmos não são independentes e não podem ser executados diretamente pelo usuário, ao contrário

do segundo. A indicação se o algoritmo pode ou não ser acionado diretamente é representada pelo atributo *steponly*.

A Figura 11 apresenta o diagrama de classes contendo *AbstractMatcher* e uma implementação do algoritmo proposto por Leme [20]. O capítulo 4 apresenta maiores detalhes sobre o desenvolvimento deste algoritmo no *Matchmaking*.

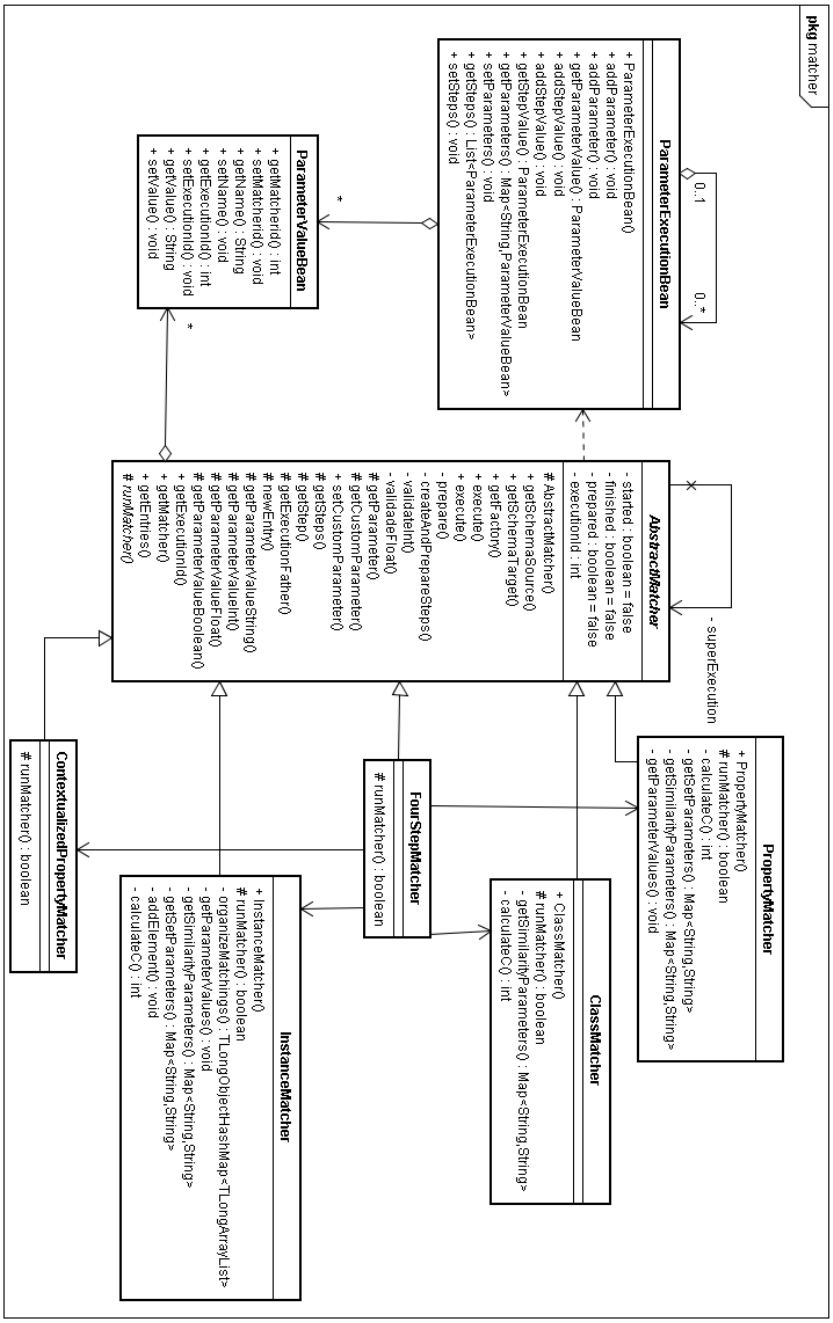

Figura 11 - Algoritmos de alinhamento[4].

[4] Parâmetros dos métodos foram omitidos.

Cada passo da implementação do algoritmo proposto por Leme (*Property Matcher*, *Class Matcher*, *Instance Matcher* e *Contextualized Property Matcher*) estende a classe *AbstractMatcher* de modo que possam ser reconhecidos como algoritmos de alinhamento. Além disso, a classe *FourStepMatcher* é a responsável por coordenar a execução dos quatro passos e informar as entradas de alinhamento finais. Nesse caso, apenas *FourStepMatcher* pode ser acionada diretamente pelo usuário.

A Figura 12 apresenta a seqüência genérica executada por todos os algoritmos implementados no *Matchmaking*. O comando *ExecuteMatcher* aciona o algoritmo escolhido informando os esquemas e *datasets* de origem e destino, os valores de parâmetros e a conexão com o banco de dados (*DAOFactory*). Os parâmetros passados precisam estar na estrutura recursiva definida pelo algoritmo para seus passos. Para isso, os valores precisam ser informados através da estrutura de dados representada pela classe *ParameterExecutionBean* e *ParameterValueBean* (Figura 11). Cada subalgoritmo precisa ter uma instância da classe *ParameterExecutionBean* que irá agregar os valores dos parâmetros desse item e ter uma lista com outros *ParameterExecutionBean* correspondentes aos subalgoritmos que porventura este *passo* venha ter.

Como primeira tarefa, o *ExecuteMatcher* inicia a preparação da execução. Esta se resume a validar os parâmetros informados (se eles estão com valores corretos segundo o tipo de dado declarado e se todos foram preenchidos), salvar a execução no banco de dados e preparar os seus subalgoritmos. No final da etapa de preparação, teremos toda a estrutura de execução pronta.

Se a etapa de preparação for concluída com sucesso. O método invoca a execução do método abstrato *runMatcher* que deve ser implementado pelo algoritmo. É nesse ponto que o processamento da similaridade acontece e o algoritmo é livre para solicitar funções de similaridade, geradores de representação de dados e passos na forma e na ordem que preferir.

Muitas vezes um passo pode precisar compartilhar informações com outro. Isso se dá através do método *setCustomParameter* que permite a passagem de objetos entre as partes.

Ao final do procedimento, o algoritmo deve acionar o método *newEntry* para cada entrada de alinhamento que o algoritmo calcular. Quando o método abstrato *runMatcher* finalizar, a informação da duração da execução será salva no

banco de dados e o processo será dado como concluído.

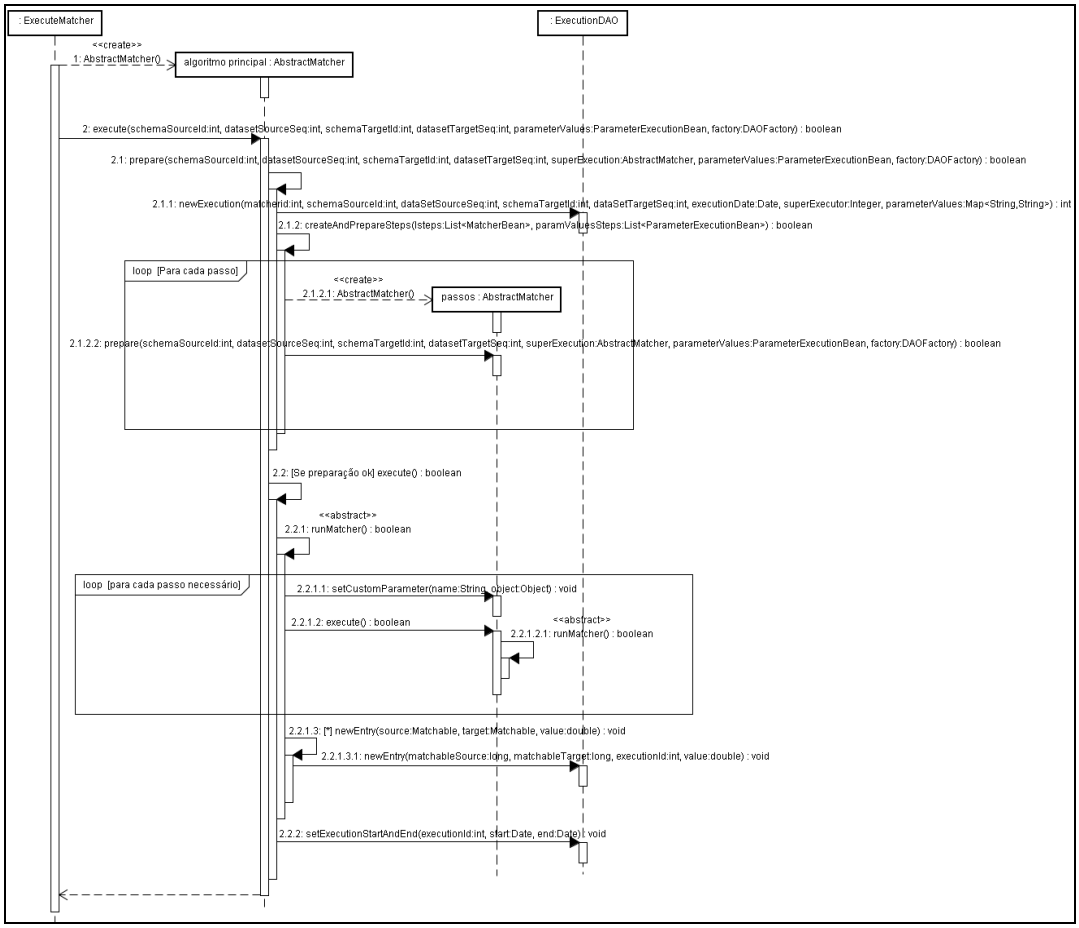

Figura 12 - Seqüência genérica de um algoritmo de alinhamento.

3.2.7. Esquema

O pacote *matchmaking.schema* (Figura 13) apresenta um conjunto de classes que representam os elementos alinháveis (*Matchable*) do esquema OWL. Basicamente, temos uma representação do esquema (*Schema*), suas classes (*Clss)*, propriedades no contexto de uma classe (*CProperty*) e instâncias (*Instance*).

Esse é o conjunto de classes utilizado pelos algoritmos de alinhamento para comparar esquemas. Apesar de conter informações sobre os elementos alinháveis, nem todas as informações do esquema estão disponíveis nesse modelo. De modo a cumprir o terceiro requisito da seção 3.2, uma integração com o *framework Jena*[5] foi criada.

Essa integração com o *Jena* foi criada da forma mais transparente possível.

[5] "Jena – A Semantic Web Framework for Java" é uma biblioteca desenvolvida em Java com o objetivo de viabilizar a construção de aplicações de Web Semântica. Ela possui um ambiente pronto para se trabalhar com arquivos OWL e inclui um motor baseado em regras de inferência. http://jena.sourceforge.net/

Basicamente, quando um algoritmo recebe uma variável do tipo *Schema* internamente ele também está recebendo um objeto *Jena* (*OntModel*) referente ao esquema (e pode ser acessado pelo método *getSchema*). Quando o algoritmo solicita, por exemplo, uma classe ao objeto *Schema*, este retorna o objeto referente a essa classe e, internamente, um objeto *Jena* (*OntClass*). O mesmo vale para propriedades (*OntProperty*) e instância (*Individual*).

Um dos problemas dessa implementação é o tempo gasto pelo *Framework* para processar todos os elementos de um esquema. Por isso, uma otimização realizada foi utilizar o recurso de carregamento preguiçoso (*lazy load*). O funcionamento é simples: os objetos só são carregados pelo *Jena* a partir da primeira solicitação. Se um algoritmo não precisar dos objetos do *framework* ele não precisará arcar com os cursos desse carregamento.

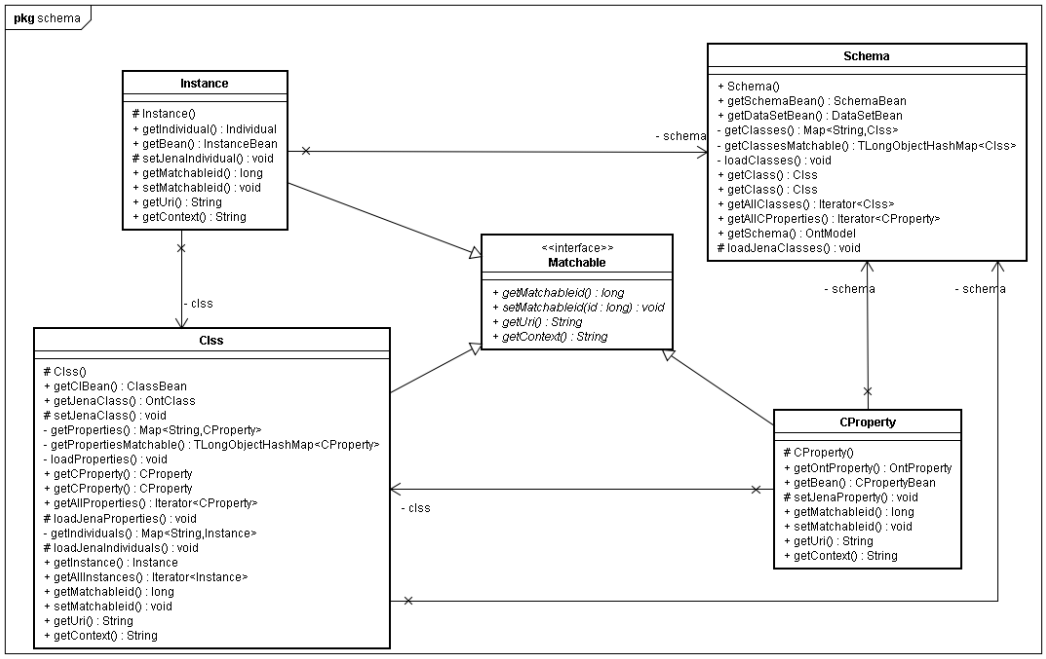

Figura 13 – esquema[6]

Uma segunda otimização foi o carregamento de partes do esquema. Ao invés de carregarmos o esquema inteiro a partir da primeira solicitação, ou carregar apenas o elemento solicitado, foi feito um carregamento a partir do tipo de objeto. Se o algoritmo solicitar o objeto *Jena* de uma classe, o sistema fará uma consulta buscando automaticamente todas as classes dele. Se ele buscar pelas instâncias, buscará todas elas. E se buscar pelas propriedades, buscará primeiro as

[6] Parâmetros dos métodos foram omitidos

classes e depois as propriedades.

Essa otimização foi desenvolvida dessa forma porque a busca de um único objeto é muito mais custosa do que uma lista de objetos semelhantes. Se um algoritmo precisa do objeto *Jena* de uma classe, é muito provável que ele irá precisar para todas as classes. O mesmo vale para instâncias e propriedades. Ao mesmo tempo, o algoritmo pode precisar apenas de todas as classes e não haveria necessidade de carregar todo o resto.

A última otimização realizada é que um esquema é reaproveitado por todos os *passos* do algoritmo. Dessa forma, se um passo já carregou todas as classes *Jena*, não será preciso fazer essa operação novamente. Ou seja, quanto mais passos existirem no algoritmo, mais rápido ele estará quanto ao carregamento das informações.

3.2.8. Banco de dados

O pacote *matchmaking.dao* é a camada responsável por todo acesso à dados do *framework*. Nessa camada, utilizamos o padrão de projeto DAO (*Data Access Object*) que tem como objetivo centralizar o acesso a dados e permitir que possamos trocar de tecnologia de armazenamento de dados sem precisar alterar o resto do código. O *Microsoft SQL Server* foi escolhido como o SGBD utilizado no sistema e, portanto, foram criados um conjunto de classes para trabalhar com este. O Apêndice B. apresenta os passos de configuração do Matchmaking e como um novo conjunto de classes de acesso a dados pode ser implementado.

A Figura 14 apresenta o diagrama de classes do pacote *matchmaking.dao* juntamente com as classes de acesso à dados desenvolvidas para o SQL Server. As classes do tipo *DAO* foram divididas de forma semelhante aos pacotes apresentados:

- *SchemaDAO*: é a classe que cuida da manipulação dos dados do pacote *matchmaking.schema* (seção 3.2.7).
- *SetDAO*: Classe responsável pela manipulação dos dados do pacote *matchmaking.sets* (seção 3.2.4), inclusive as informações de proveniência.

- *SimilarityDAO*: responsável pelo controle dos dados do pacote *matchmaking.similarity* (seção 3.2.5), inclusive as informações de proveniência.
- *MatcherDAO:* é a classe que realiza as operações de inclusão, alteração e recuperação dos dados de um algoritmo (maiores detalhes na seção 3.2.6).
- *ExecutionDAO:* classe que gerencia os dados relativos à execução de um algoritmo, inclusive as informações de proveniência (maiores detalhes na seção 3.2.6).

As Figura 15 e Figura 16 ilustram o modelo físico do banco de dados utilizado pela ferramenta. Esse modelo é derivado do conceitual apresentado na Figura 2 e apresenta uma proposta de como os dados devem ser armazenados para atender todos os requisitos da ferramenta. É importante salientar que, da forma como a camada de acesso a dados foi desenvolvida, outras soluções de modelos podem ser criadas e, inclusive, pode-se utilizar meios alternativos de armazenamento dos dados (como por arquivos). Tudo depende de como otimizar as consultas para tornar a ferramenta mais eficiente, se possível.

O próximo capítulo apresenta testes com a ferramenta, realizados através de uma interface gráfica e utilizando o algoritmo de alinhamento de esquemas proposto por Leme.

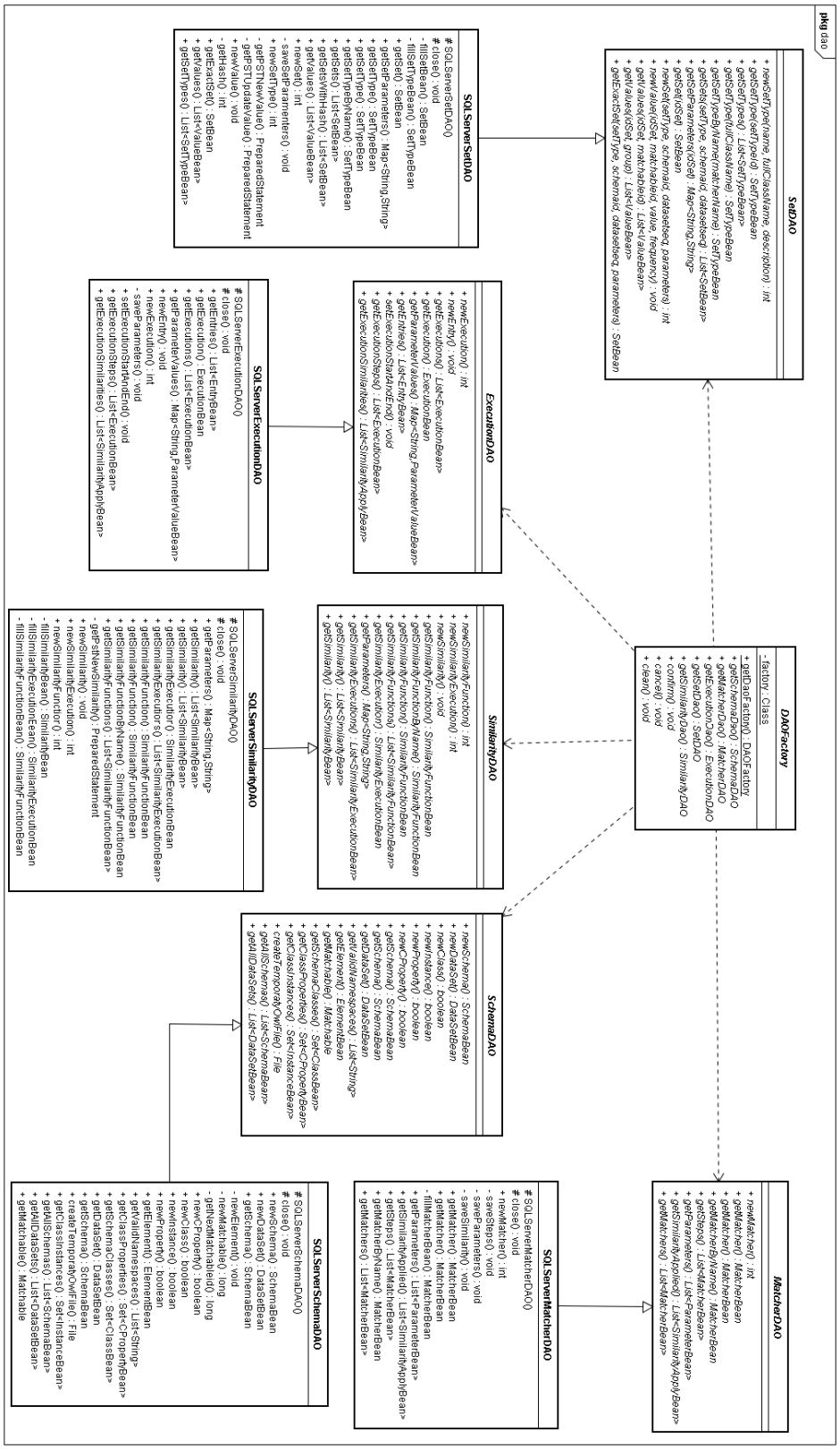

Figura 14 - DAO[7]

[7] Atributos e parâmetros dos métodos foram omitidos.

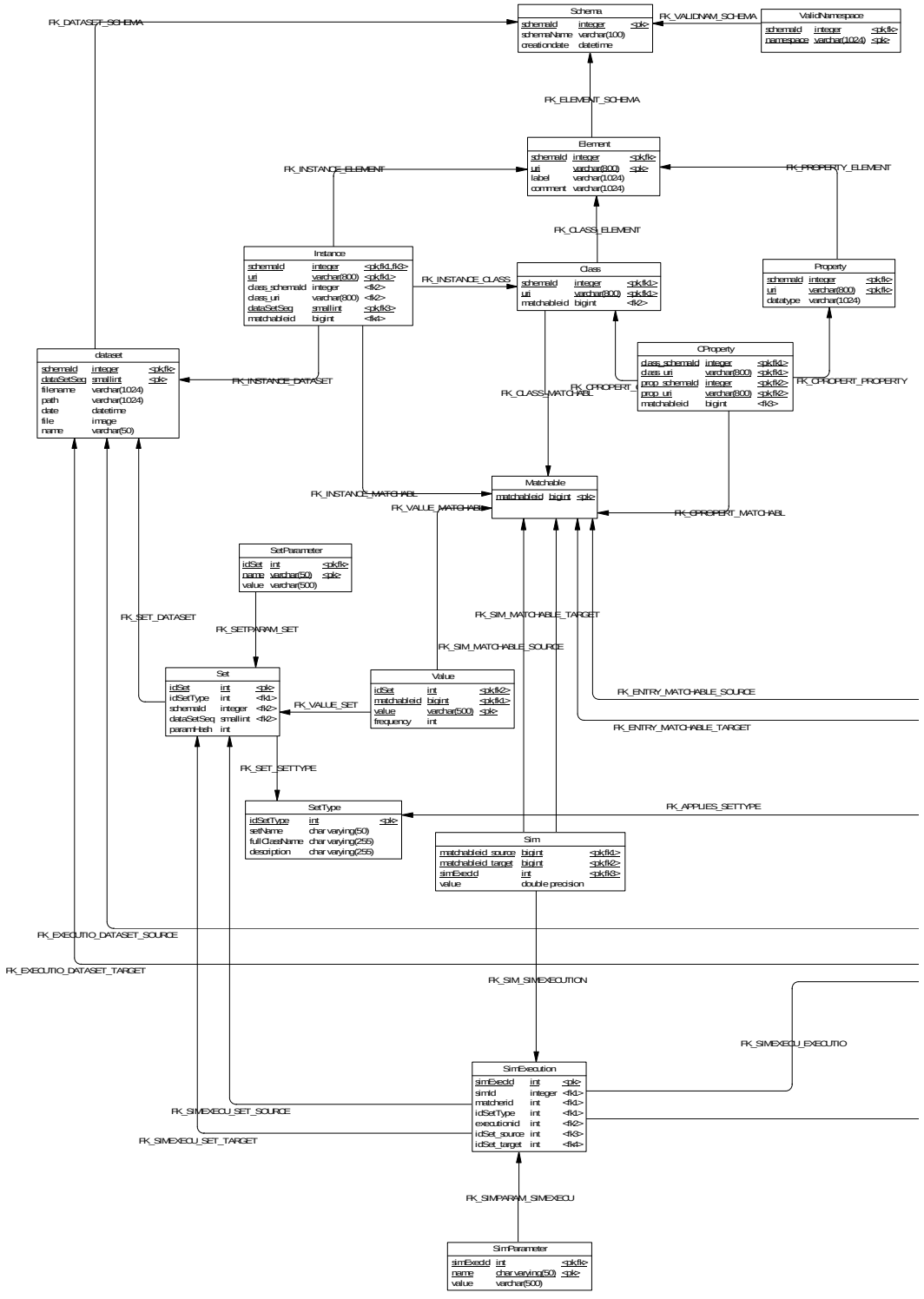

Figura 15 - Modelo físico do banco de dados (1).

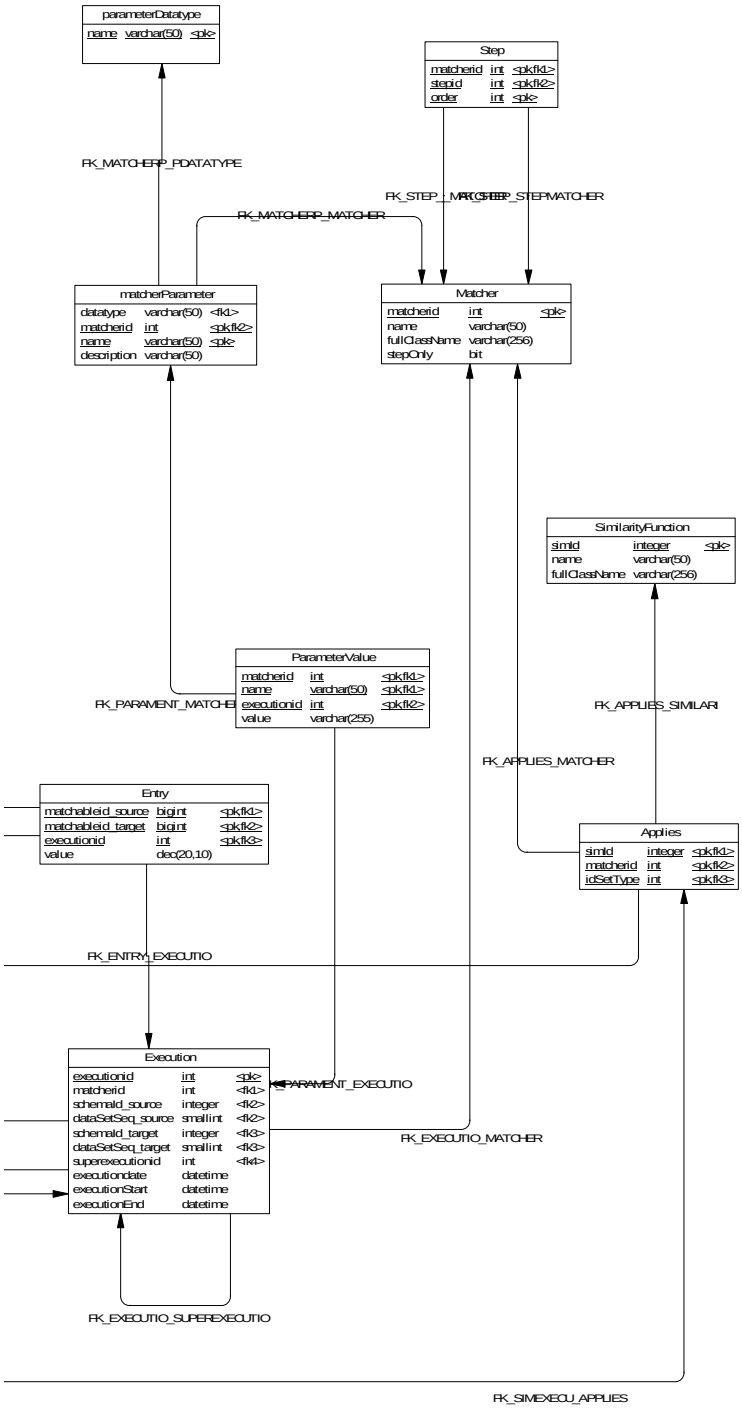

Figura 16 - Modelo físico do banco de dados (2).