7
O *Framework* e seu Sistema Multi-Agentes

A decisão de adotar uma visão de sistema multi-agentes para o desenvolvimento do *framework* foi devida à melhor compreensão do problema adotado caso ele fosse visto dessa maneira - um agente classificador pode ser encarado como uma entidade que fica constantemente buscando documentos, selecionando os que devem ser processados, processando os documentos selecionados segundo algum algoritmo e reportando os resultados obtidos a alguém que possa continuar o processamento (realizar o que tiver que ser realizado com os dados processados).

O sistema foi desenvolvido utilizando o paradigma de orientação a objetos, sendo que Java foi utilizada como linguagem de programação. É interessante discutirmos dois pontos em especial:

- A estrutura interna dos agentes;
- O sistema multi-agentes que compõe o *framework*.

Esta separação foi adotada a fim de evitar que questões relativas a um agente em especial se misturassem com questões relativas ao sistema como um todo [50, 51]. A Figura 1 ilustra esta preocupação, onde é representada a camada de gerenciamento dos agentes. Além disso, um dos objetivos era deixar o sistema multi-agentes o mais escondido possível, de forma que o instanciador do *framework* tivesse o menor número de pontos de contato com os agentes. Preferencialmente, o instanciador nem precisa saber que trata-se de um sistema multi-agentes, ou mesmo que há agentes de software a serem executados.

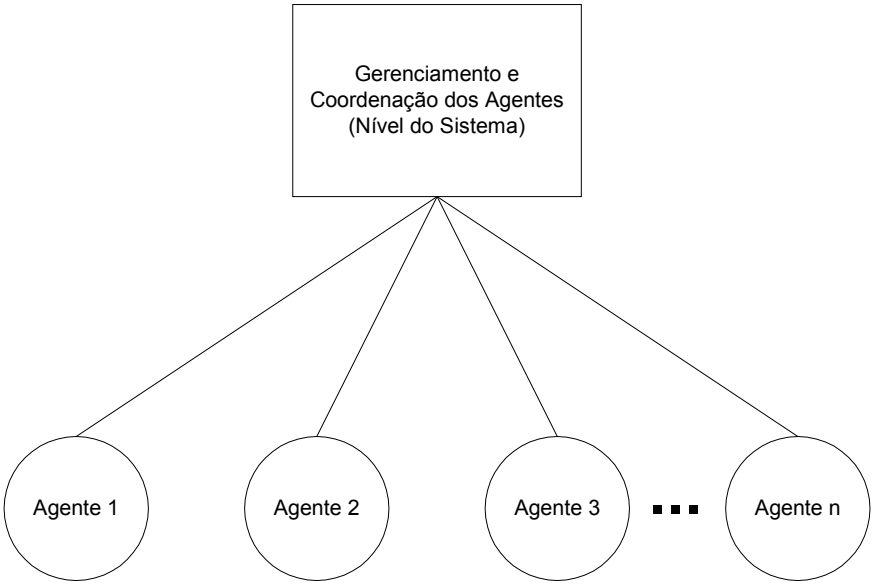

Figura 1: Visão geral do sistema multi-agentes

7.1
O Agente de Software

7.1.1.
Questões de implementação

O agente de software é composto por seis subsistemas. Alguns deles podem ser compartilhados. A Figura 2 ilustra a arquitetura interna de um agente, com todos os seis subsistemas. Ela é resultante da expansão de um único agente de software da Figura 1 (representado por um círculo).

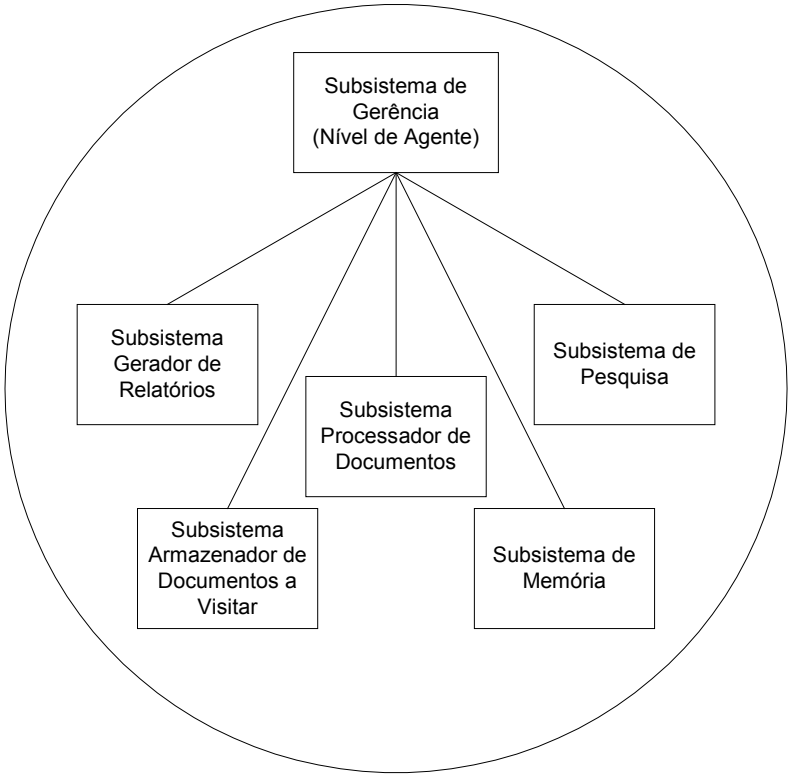

Figura 2: Estrutura interna de um agente

Cada um dos subsistemas é responsável por um aspecto específico:

- **Subsistema Processador de Documentos:** É responsável por realizar o processamento dos documentos a serem classificados, extraindo as informações relevantes como seu texto puro (no caso de documentos que tenham informações de formatação, como um HTML ou PDF), e também novos documentos que sejam referenciados (no caso de documentos que tenham informações de *hyperlinks*, como um HTML ou PDF).
- **Subsistema Armazenador de Documentos a Visitar:** É responsável por analisar e armazenar caminhos para documentos que ainda serão visitados durante a classificação. No início do processo de classificação, este subsistema deve receber os documentos a serem classificados *a priori*.

- Este subsistema pode ser alimentado também pelo próprio agente de software, caso os documentos classificados sejam capazes de indicar novos documentos – neste caso, para evitar que a busca torne-se excessivamente grande, há o conceito de altura do documento, que funciona como um corte na árvore de busca: todos os documentos inseridos inicialmente no subsistema recebem uma altura *h*. Quando um documento é processado e gera novos documentos a serem processados, tais documentos recebem a altura ($h - 1$). O processo se repete até que *h* seja zero. Quando um documento de altura zero é processado, não se verifica se ele indica novos documentos.

- O motivo de deixar a responsabilidade de armazenamento separada em um subsistema é possibilitar que seja feita uma crítica inicial em relação aos documentos que ainda serão processados – por exemplo, é possível inserir prioridades a documentos, ou ainda inserir filtros para evitar que uma classe de documentos seja processada.

- **Subsistema de Memória**: É o subsistema responsável por guardar, de forma permanente, a lista dos documentos já classificados (e não alterados desde a última classificação), evitando que sejam reprocessados inutilmente. Porém, há casos em que é desejável que sejam re-classificados alguns documentos que podem dar origem a outros ainda não classificados. Um exemplo típico deste caso é um documento no formato HTML já processado, mas que contenha *hyperlinks* para outros documentos ainda não processados. Desta forma, existe um outro conceito atrelado à memória, que é a altura de corte, que será discutida posteriormente.

- **Subsistema de Pesquisa:** É o subsistema que encapsula o algoritmo utilizado para classificar os documentos.

- **Subsistema Gerador de Relatórios:** A forma como os resultados obtidos a partir da classificação são utilizados é tratada por este subsistema, que é claramente um dos *hot spots* do *framework*. Por exemplo, uma possível instanciação seria uma aplicação de classificação

de documentos HTML para a *web*, na qual podem ser dadas duas opções aos usuários: receber o resultado da classificação por e-mail (ideal para grandes coleções de documentos, cuja classificação levará muito tempo), ou receber os resultados em seu navegador, durante a execução (ideal para pequenas coleções de documentos, cuja classificação levará pouco tempo). Outro exemplo de instanciação seria uma aplicação de documentos em uma *intranet*, cujo resultado fosse colocado em uma base de dados, que seria posteriormente usada num portal de conhecimento interno.

- **Subsistema de Gerência**: É o "cérebro" do agente, sendo responsável por coordenar os demais subsistemas. Este subsistema é a parte ativa do agente, representando um *Thread* independente de execução.
- Analisaremos agora, com mais detalhes, cada um dos subsistemas.

7.1.1.1 Subsistema Processador de Documentos

Conforme mostrado na Figura 3, este subsistema é composto por três classes: IFabricaDocumentos, IDocumento e ClDocumentoAVisitar ("I" significa interface, e "Cl", classe). IDocumento é uma interface que deve ser implementada por um documento processado pelo sistema, e define os métodos que o *framework* deve utilizar para obter as informações necessárias para classificá-lo. Essa interface tem por objetivo encapsular todas as responsabilidades do processamento de um documento, tornando a abertura de um documento totalmente transparente para o resto do sistema. Documentos são instanciados a partir de instâncias da classe ClDocumentoAVisitar. Esses objetos representam documentos que ainda não foram processados pelo Subsistema Processador de Documentos, e contêm todas as informações necessárias para que o documento seja localizado e aberto corretamente. A instanciação de documentos é tarefa das classes que implementam IFabricaDocumentos. A dupla (IFabricaDocumentos, IDocumento) é uma implementação do *AbstractFactory pattern* [52].

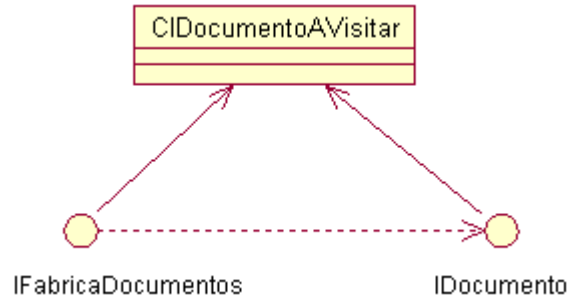

Figura 3: Diagrama de classes do Subsistema Processador de Documentos

A classe ClDocumentoAVisitar possui quatro métodos:

- `int obterAltura( )` : retorna a altura em que esse documento foi encontrado. Conforme já mencionado anteriormente, esse conceito só faz sentido no caso de documentos que possam indicar o caminho de outros documentos (como documentos HTML, por exemplo).
- `int obterAlturaCorte( )`: retorna a altura de corte do documento. Esse conceito será discutido posteriormente, na descrição do subsistema de memória.
- `String obterCaminho( )`: retorna uma String representando o caminho do documento a ser processado. Optou-se por deixar a representação de caminhos em Strings devido à bem-sucedida experiência com as *Uniform Resource Identifiers* (ou URIs) [53], que são o padrão para identificação de recursos na *web* (imagens, documentos, serviços, e-mails, etc).
- `String obterGrupo( )`: retorna uma String contendo o grupo do documento. O conceito de grupo vem do fato que, durante uma classificação, diferentes locais contendo diferentes documentos podem ser processados simultaneamente, e pode ser interessante manter uma diferenciação entre documentos oriundos de cada local. Por exemplo, uma classificação pode ser efetuada numa intranet, buscando por documentos em diferentes máquinas, e cada máquina poderia ser definida como um grupo diferente. Mesmo se a classificação ocorresse

numa única máquina, diferentes diretórios podem ser definidos como diferentes grupos.

- `int hashCode()`: retorna um inteiro de 4 bytes obtido através de operações matemáticas realizadas com o caminho do documento. Esse valor é utilizado pelo subsistema de memória, que será discutido posteriormente.

A interface IFabricaDocumentos possui apenas um método, de assinatura `IDocumento processarDocumento( int iIDScanner , ClDocumentoAVisitar DocumentoAVisitar )`, que recebe como parâmetros o documento a ser processado e o identificador do agente que está solicitando o processamento, e retorna uma instância de alguma classe que implemente a interface IDocumento.

A interface IDocumento possui os seguintes métodos:

- `ClDocumentoAVisitar obterProximoDocumento()` : Este método é o responsável por retornar outros locais que contenham documentos indicados por este documento. Funciona como uma instância do *pattern Iterator* [52].
- `procurarTexto( String sTexto )`: Retorna o número de vezes que uma String ocorre no documento.
- `String toString()`: Retorna uma representação em String desse documento.
- `int hashCode()`: retorna um inteiro de 4 bytes obtido através de operações matemáticas realizadas com texto do documento. Esse valor é utilizado pelo subsistema de memória, que será discutido posteriormente.
- `ClDocumentoAVisitar obterDocumentoAVisitar()`: Retorna a instância da classe ClDocumentoAVisitar que foi utilizada para geração desse documento.

- `String obterTitulo()`: Retorna o título desse documento. Caso este seja desconhecido, é deixado a cargo do instanciador o valor retornado (por exemplo, pode retornar um valor padrão ou `null`).
- `String[] obterResenhas( String sTexto , int iCaracteres , int iNumResenhas )`: Retorna os trechos do texto do documento onde a String sTexto ocorre. Essas resenhas serão montadas da seguinte forma: serão selecionados iCaracteres à esquerda de sTexto e iCaracteres à direita de sTexto. A seguir, é feito um processamento para evitar que as palavras que iniciam e terminam a resenha sejam cortadas pelo meio. São fornecidas somente as iNumResenhas resenhas, ou seja, se a palavra ocorrer no documento mais vezes, então algumas resenhas serão descartadas (a decisão sobre qual resenha descartar fica a cargo do instanciador).
- `String[] obterResenhas( String sTexto , int )`: Igual em funcionalidade ao método anterior, exceto que não há limite na quantidade de resenhas a serem entregues – serão entregues todas as resenhas encontradas.
- `Collection obterParametro( String sParametro )`: Retorna os valores atrelados a um parâmetro. A idéia desse método é fornecer uma funcionalidade idêntica à da classe Properties da API Java [54], fornecendo uma maneira do instanciador atrelar mais parâmetros a classes que implementem IDocumento além dos já definidos pelos métodos da classe.

7.1.1.2 Subsistema Armazenador de Documentos a Visitar

Este subsistema tem por objetivo encapsular todas as responsabilidades relativas à seleção e ao armazenamento dos documentos *que ainda serão processados*. Quando o Subsistema Processador de Documentos termina o processamento de um documento, este é logo em seguida analisado e verificado quanto à sua capacidade de indicar novos locais onde outros documentos possam

ser encontrados (por exemplo, um documento no formato HTML pode conter *hyperlinks* para novos documentos HTML). Cada um desses locais dá origem a uma instância da classe ClDocumentoAVisitar, que será adicionada ao Subsistema Armazenador de Documentos a Visitar. Cabe a esse subsistema criticar o local representado a fim de decidir se deve ou não ser visitado; ou ainda, impor uma ordem de prioridade na visita – trata-se de um *hot spot*, ficando a cargo do instanciador do *framework*.

Duas classes compõem esse subsistema: IArmazenamentoStrategy (uma interface) e a já mencionada ClDocumentoAVisitar. A interface IArmazenamentoStrategy define um "armazém de locais a serem visitados", uma vez que todos os locais são nela armazenados. Cabe ao instanciador implementá-la de acordo com suas necessidades. Essa interface é um *Strategy pattern* [52]. O diagrama de classes completo está na Figura 4.

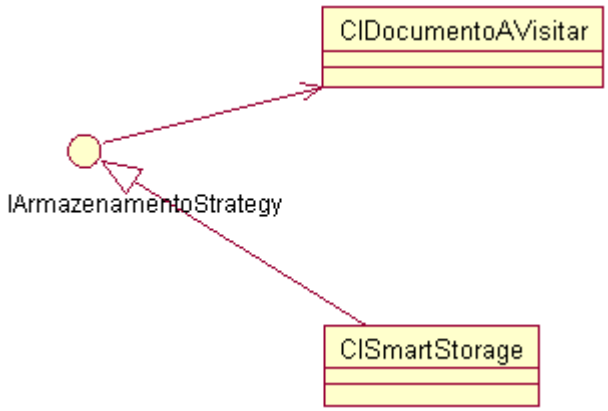

Figura 4: Diagrama de classes do Subsistema Armazenador de Documentos a Visitar

A interface IArmazenamentoStrategy contém os seguintes métodos:

- `public boolean isEmpty( int iIDScanner )`: Verifica se ainda há locais com documentos a serem visitados.
- `public void clear( int iIDScanner )`: Remove todos os documentos a serem visitados, fazendo com que a lista fique vazia.
- `public int getSize( int iIDScanner )`: Retorna o tamanho instantâneo da lista.

- `public int getTotalSize( )`: Retorna o número de documentos que já estiveram presentes nessa lista.
- `public void add( ClDocumentoAVisitar DocumentoAVisitar , int iIDScanner )`: Adiciona um documento à lista.
- `public ClDocumentoAVisitar get( int iIDScanner )`: Obtém um documento da lista (e o remove dela).

Durante o desenvolvimento do trabalho, particularmente nas instanciações, foi possível perceber que um bom filtro de documentos consiste em verificar se o seu caminho continha um determinado padrão de caracteres, e de acordo com isso, aceitá-lo ou não.

Por exemplo, se estamos processando documentos HTML de um *website* específico, então os objetivos principais do filtro são:

- Evitar que a ferramenta saia do *website*, ou seja, evitar que a ferramenta escape para algum outro domínio devido à existência de *hyperlinks* – por exemplo, ao classificarmos os documentos de O Globo, evitar que a ferramenta chegue ao *website* da revista Época devido à existência de *hyperlinks*.
- Evitar que a ferramenta visite documentos dentro do próprio *website* que sabidamente não devem ser processados – por exemplo, no caso do jornal O Globo, evitar que seja processada a seção de Classificados.

Porém, se estamos processando documentos do tipo texto, o domínio de busca poderia ser uma pasta (diretório), que por sua vez teria sub-pastas. Algumas dessas sub-pastas podem não ser de interesse, logo deveria haver uma forma de excluí-las. Da mesma forma, poderíamos estar interessados em apenas processar documentos com uma certa lei de formação em seu nome (por exemplo, somente documentos com um determinado prefixo ou sufixo, segundo alguma nomenclatura). Assim, teríamos os mesmos objetivos mencionados anteriormente:

- Evitar que a ferramenta entre em diretórios indesejados, o que está diretamente relacionado com o primeiro objetivo dos documentos HTML;
- Evitar que a ferramenta processe documentos nas pastas que sabidamente não devem ser processadas, o que está diretamente relacionado com o segundo objetivo dos documentos HTML.

Devido a esse uso comum, o *framework* já possui uma instanciação desse subsistema que contempla esses dois objetivos: é a classe ClSmartStorage, que implementa a interface IArmazenamentoStrategy, recebendo em seu construtor uma lista de Strings que "definem" um grupo de documentos, bem como uma lista que "exclui" documentos. Estas listas são confrontadas com o caminho dos documentos encontrados de forma a decidir se um novo documento deve ou não ser processado.

7.1.1.3 Subsistema de Memória

As responsabilidades deste subsistema consistem no armazenamento permanente de informações acerca dos documentos classificados, a fim de que seja possível evitar que os mesmos sejam re-classificados desnecessariamente.

Há dois tipos de memória: a memória volátil, válida somente durante uma pesquisa, e a memória permanente, que é válida mesmo em pesquisas diferentes. A volátil tem por objetivo evitar que um documento seja re-classificado numa mesma pesquisa (detecção de duplicatas), ao passo que a permanente evita que um documento seja re-classificado em pesquisas diferentes. A separação da memória em duas partes teve como principal objetivo aumentar o domínio de problemas que podem ser resolvidos pelo *framework* proposto. Ocorre que, em determinadas aplicações de classificação, é necessário que os documentos sejam re-classificados totalmente após cada execução (por exemplo, aplicações de busca direta em texto, sem pré-processamento) ao passo que em outras isto pode (e deve) ser evitado, para fins de aumento de eficiência (por exemplo, numa aplicação de classificação de documentos de uma organização para disponibilização em uma intranet).

Devido a esta flexibilidade do domínio de problemas atacado, essa parte do subsistema constitui outro *hot spot* do *framework*. Já a memória volátil não é flexível, sendo um *frozen spot*.

Esse subsistema é formado por quatros classes: ClDocumentosVisitados, que trata da memória volátil (sendo, portanto, uma classe concreta), a IPagesKeeper, que trata da memória permanente (uma interface) e as classes IDocumento e ClDocumentoAVisitar, ambas já analisadas anteriormente no subsistema processador de documentos.

A representação interna da memória volátil é bastante simples:

- Para cada ClDocumentoAVisitar processado, obtém-se seu código hash através do método `hashcode`. Esse inteiro é chamado de *hash volátil*, e a função de cálculo utilizada, que é denominada função de hash, é a que já vem com a classe String de Java, cuja documentação pode ser consultada na API [54]. Esta função garante que a distribuição dos valores entre as possíveis Strings dificulta que haja coincidência de hashs.
- Os hashs voláteis obtidos são então armazenados em uma lista ordenada.

A memória permanente, por se tratar de um ponto de flexibilização, não está implementada. Porém, uma possibilidade a ser considerada na implementação dessa parte é fazer uso de uma estratégia semelhante a da memória volátil:

- Para cada IDocumento processado, obtém-se seu código de hash através do método `hashcode` já mencionado anteriormente. Esse método já é implementado na classe Object de Java, porém ele é declarado explicitamente como abstrato na interface a fim de deixar claro ao instanciador que é altamente recomendável reescrevê-lo.
- Os hashs obtidos são então armazenados de forma permanente, ficando esta tarefa a cargo do instanciador.

A decisão de armazenar hashs (e não Strings) de forma ordenada possibilita que, se em um dado momento houver n elementos na lista, uma consulta ou inserção de elemento na lista terá performance proporcional a *log(n)* se

utilizarmos o algoritmo de busca binária (pois uma comparação entre inteiros pode ser feita em tempo constante), o que garante a eficiência desse processo. Outro ponto que merece destaque é que os hashs possuem seu tamanho limitado a quatro bytes, o que não ocorre com as Strings, que representam os locais dos documentos, que podem ter dezenas de bytes.

A classe ClDocumentosVisitados possui dois métodos que merecem destaque:

- `add( Object Objeto )` : Adiciona um documento à memória volátil
- `contains( Object Objeto )` : Verifica se um documento pertence à memória volátil

A interface IPagesKeeper contém três métodos:

- `add( IDocumento Documento )` : Adiciona um documento à base de documentos já classificados.
- `contains( IDocumento Documento )` : Verifica se um documento pertence à base (retorna `true` ou `false`).
- `close()` : Fecha a memória para uso (ou seja, após a chamada desse método, não é permitida a chamada dos métodos `add` ou `contains`).

Para desativar a memória permanente, basta escrever uma classe cuja implementação dos métodos `add` e `close` sejam vazias (ou seja, não executem código nenhum) e cuja implementação do método `contains` sempre retorne `false`. Devido à simplicidade desta classe, o próprio *framework* já possui essa instanciação na classe ClDummyKeeper.

O diagrama de classes desse subsistema pode ser visto na Figura 5.

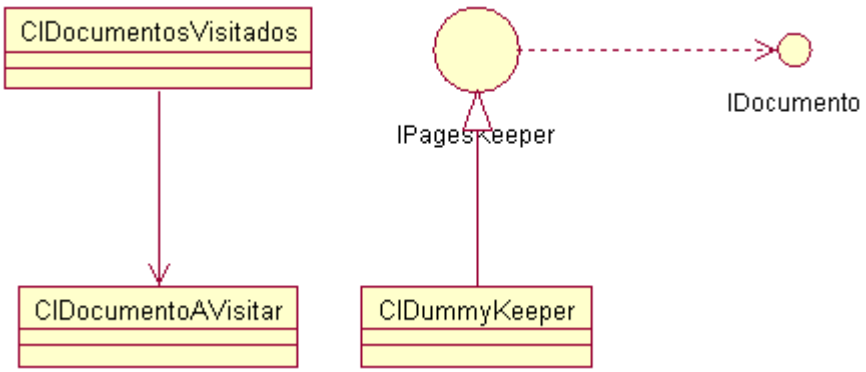

Figura 5: Diagrama de classes do Subsistema de Memória

7.1.1.4 Subsistema de Pesquisa

Esse subsistema é o responsável por encapsular o algoritmo de classificação utilizado. É formado por duas interfaces e uma classe abstrata: IPesquisa, IDocumento e ClRespostaPesquisa.

IPesquisa é o grande encapsulador do algoritmo. Esta classe deve conter todas as etapas do algoritmo e ser capaz de responder como ele classifica o documento. Em outras palavras, a classe que implementar IPesquisa contém o código do algoritmo. Essa parte deve ser implementada utilizando os métodos definidos, que são três:

- `toString()`: deve retornar uma representação em forma de `String` da pesquisa. Pode ter diversas utilizações, mas seu maior uso (em tempo de desenvolvimento) foi para fins de correção de erros (*debug*).
- `ocorrerPesquisa( IDocumento Documento )`: retorna um booleano indicando se a aplicação da pesquisa ao documento tem ou não sucesso .
- `pesquisar( IDocumento Documento )` : retorna uma instância da classe ClRespostaPesquisa, contendo informações sobre a aplicação da pesquisa ao documento.

A diferença entre os métodos `ocorrerPesquisa` e `pesquisar` está no grau de detalhe que se deseja na resposta. O objetivo do primeiro é retornar mais informações a respeito da pesquisa que foi realizada. Essas informações variam de instância para instância (daí o fato de ClRespostaPesquisa ser abstrata), mas podemos citar duas que seriam interessantes para classificação de documentos extensos ou pertinentes a várias classes:

- ***Grau de certeza da ocorrência***: pode ser indicado o grau de certeza sobre a classificação do documento.
- ***Trecho do documento que mais se aproxima do desejado***: pode indicar o(s) trecho(s) do documento que contém as informações que levaram à decisão de como classificá-lo (especialmente útil para documentos muito extensos, que possam ser incluídos em diversas categorias).

Apesar de ser abstrata, a classe ClRespostaPesquisa não possui nenhum método abstrato. Os métodos que essa classe possui são os seguintes:

- `IPesquisa obterPesquisa()` : Obtém a pesquisa que gerou essa resposta.
- `ClDocumentoAVisitar obterDocumentoAVisitar()` : Obtém o caminho do documento que gerou essa resposta.

O diagrama de classes desse subsistema pode ser visto na Figura 6.

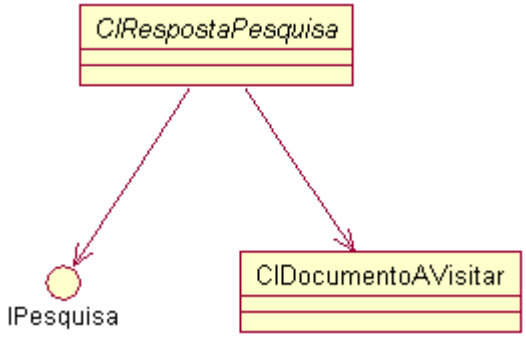

Figura 6: Diagrama de classes do Subsistema de Pesquisa

7.1.1.5
Subsistema Gerador de Relatórios

Esse subsistema é responsável por receber os resultados obtidos do processamento dos documentos – a forma como os resultados são utilizados também é um *hot spot* do *framework*.

Há duas classes compondo esse subsistema: ISender e ClRespostaPesquisa. ISender é uma interface que age como um *Observer* [52] sendo avisada sempre que um resultado de classificação estiver disponível. Possui três métodos, descritos a seguir:

- `void addToAnswer( ClRespostaPesquisa Answer )` : É utilizado para reportar que uma classificação de um documento foi satisfatoriamente efetuada.
- `void logProcess( int iAltura , String sCaminho )` : É utilizado para reportar andamento da execução do processamento da classificação – como o processo de classificação pode ser demorado, no caso em que os usuários fiquem esperando pelo resultado final da classificação, este método pode ser utilizado para avisar periodicamente o que está sendo efetuado pelo classificador.
- `void logError( String sError )` : É utilizado para reportar algum erro que ocorra durante a classificação.

ClRespostaPesquisa já foi analisada anteriormente, no subsistema de pesquisa.

Durante o desenvolvimento do trabalho, particularmente nas instanciações, foi possível perceber que dois comportamentos poderiam ser comumente desejados pelas classes que implementassem ISender:

- ***Múltiplos receptores*:** Apesar do *framework* ter sido projetado para trabalhar com apenas um ISender, em alguns casos pode ser interessante ter vários Senders (ou seja, vários receptores) recebendo o resultado das classificações. Por exemplo, poderíamos considerar o caso em que os resultados seriam armazenados em diversas bases de dados com

formatos diferentes, a fim de possibilitar integração com diferentes ferramentas de gestão de conhecimento. Nesse caso, cada base de dados teria o seu Sender, que receberia os parâmetros numa instância da classe ClRespostaPesquisa e os trataria da forma que fosse conveniente.

- ***Seleção dos resultados*:** Uma vez que foram possibilitados múltiplos receptores, é interessante ter também uma forma de que cada receptor possa selecionar quais resultados de classificação lhe são interessantes, como numa espécie de filtro. Por exemplo:
    - Podemos considerar uma aplicação de classificação por caráter de importância, em que documentos classificados como confidenciais não sejam armazenados na mesma base de dados em que documentos de outras classes são. Dessa forma, poderia haver dois Senders, um para documentos confidenciais e outro para documentos não-confidenciais, e seria necessário que cada um fosse capaz de avaliar os resultados da classificação de forma a saber como proceder.
    - Podemos considerar uma aplicação de classificação por conteúdo onde documentos seriam classificados como sendo de um ramo como economia, política, etc. em que documentos oriundos de determinadas pastas sejam armazenados em um lugar diferente. Por exemplo, documentos confidenciais poderiam ser armazenados em uma pasta especial, num cenário parecido com o apresentado no exemplo anterior. A solução é parecida, mas com ambos os Senders analisando não o resultado da classificação, mas sim a pasta de origem de cada um.

Para isso, foram criadas duas classes já no *framework*, que dão suporte às funcionalidades mencionadas:

- ClGroupSender, escrita como um *Composite pattern* [52], que recebe em seu construtor uma lista de ISenders e que tem a implementação de cada um de seus métodos simplesmente varrendo a lista de Senders e invocando o método de mesma assinatura de cada um.

- ClSelectiveSender, que fornece suporte à seleção de documentos de acordo com sua origem somente (ou seja, não aborda a questão do primeiro exemplo). Em seu construtor, ela recebe uma lista de grupos de documentos e um ISender que deve ser avisado caso a resposta recebida seja interessante. A cada invocação do método `addToAnswer`, esta classe verifica se o grupo do documento classificado (que pode ser obtido através do método `obterGrupo` do ClDocumentoAVisitar que pode ser obtido através do método `obterDocumentoAVisitar` da classe ClRespostaPesquisa) está dentre os desejados. Caso esteja, ela avisa o ISender que está encapsulando através do mesmo método `addToAnswer`.

Esse é mais um ponto onde é aplicado o conceito de *separation of concerns* [49]. O diagrama de classes do subsistema pode ser visto na Figura 7.

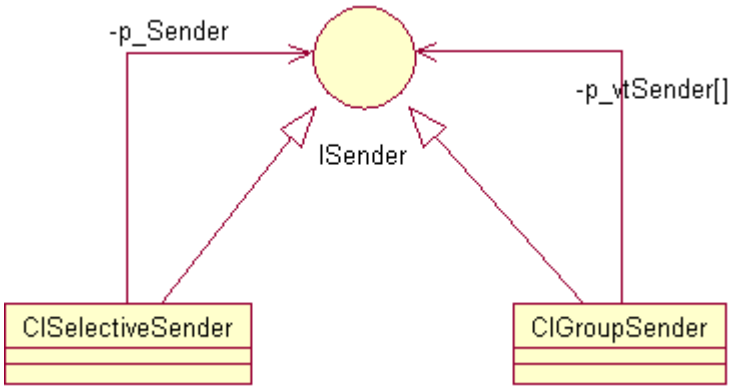

Figura 7: Diagrama de classes do Subsistema Gerador de Relatórios

7.1.1.6 Subsistema de Gerência

Este subsistema é o *frozen spot* do agente. É o responsável por coordenar a busca, gerenciando todos os demais subsistemas já mencionados anteriormente. Cada agente possui o seu próprio *thread* de execução, e contém:

- Uma lista de pesquisas (instâncias de classes que implementam IPesquisa) que devem ser aplicadas aos documentos;

- Os caminhos dos documentos a categorizar, presentes numa lista de documentos a visitar (uma instância de uma classe que implemente IArmazenamentoStrategy).
- O subsistema responsável por processar os documentos, deles extraindo as informações relevantes à classificação (uma instância de uma classe que implemente IFabricaDocumentos).
- O formatador dos resultados obtidos, a quem serão reportados todos os resultados de uma classificação (uma instância de uma classe que implemente ISender).

Um agente do sistema realiza continuamente as seguintes tarefas, nessa ordem:

- Obtém o caminho para o próximo documento a ser visitado, que está presente numa instância da classe ClDocumentoAVisitar do subsistema armazenador de documentos a visitar;
- Solicita ao subsistema processador de documentos que manipule encontre o documento e o processe;
- O documento processado é verificado para saber se indica locais de novos documentos. Caso indique, todos são repassados ao subsistema armazenador de documentos a visitar, para que este os analise e os armazene para futura visitação, se forem considerados relevantes;
- O documento processado é repassado para o subsistema de pesquisa, que aplicará o algoritmo implementado e retornará uma resposta;
- A resposta será repassada ao subsistema gerador de relatórios

A Figura 8 mostra a estrutura de um agente classificador.

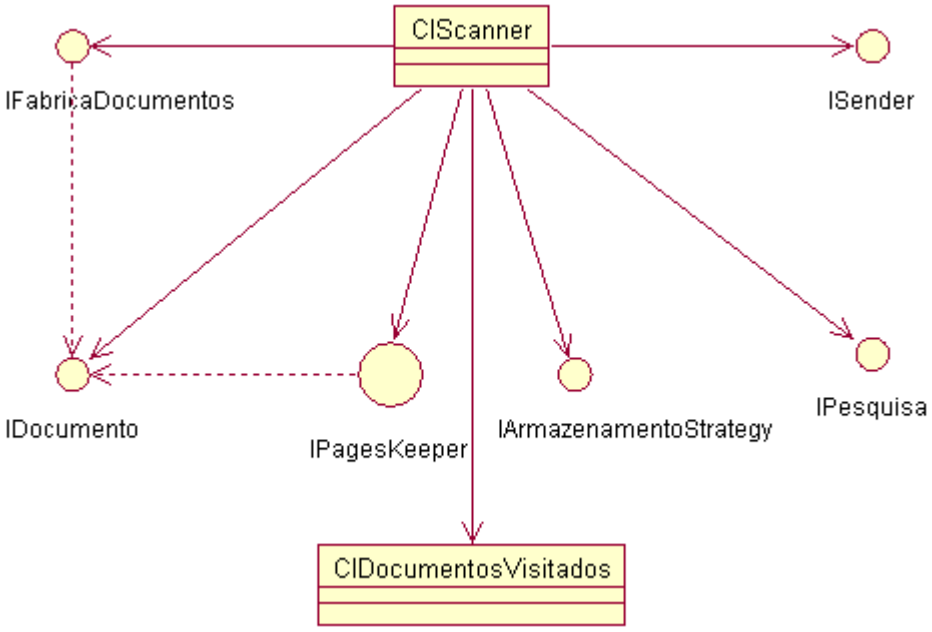

Figura 8: Estrutura interna de um dos agentes classificadores

É possível notar que a única classe componente desse sistema é a ClScanner, que encapsula todos os demais subsistemas. Essa classe é um *Thread*, logo estende a classe Thread de Java [54], e possui o método `run()`. Além disso, possui o método público `parar()`, que deve ser invocado para encerrar a classificação prematuramente, sem que todos os documentos tenham sido processados.

7.1.2
Questões de Comunicação e Coordenação

Há uma questão bastante relevante que deve ser considerada: apesar dos agentes de software serem puramente reativos e requererem um baixo grau de coordenação devido ao fato de que todos realizam tarefas de forma independente (uma vez que um agente obtém um caminho para um documento a ser visitado, ele depende somente de si mesmo para processar o documento), foi necessário modelar algum tipo de comunicação, ainda que bastante simples. Isto porque, quando um agente processa um documento, ele tem duas coisas a fazer que são relevantes para os demais agentes:

- Repassar ao **subsistema de armazenador de documentos a visitar** todas as indicações de novos documentos encontradas a partir do documento processado;
- Inserir o documento no **subsistema de memória**, para que este seja incluído tanto na memória permanente como na memória volátil.

Ambos os sistemas mantém listas para armazenar suas informações. Porém, é necessário que essas informações sejam compartilhadas entre todos os agentes – na verdade, esses subsistemas utilizam a arquitetura de *blackboards* para disponibilizar suas informações [51].

7.2 O Sistema Multi-Agentes

7.2.1 Questões de Implementação

Essa parte do *framework* funciona como um *Facade* [52] responsável por prover uma interface única de acesso ao sistema. É composta por apenas uma classe, a ClSearchManager, cujas responsabilidades estão relacionadas à criação dos agentes de busca e também de sua execução, que são:

- Iniciar a execução dos agentes com os parâmetros necessários;
- Parar a execução dos agentes prematuramente, caso seja necessário.

Há apenas dois métodos públicos nessa classe, que são os seguintes:

- void realizarPesquisa() : Dispara a pesquisa.
- void pararPesquisa() : Pára a pesquisa prematuramente.

Para gerar o sistema multi-agentes, todos os parâmetros necessários (Fábrica de Documentos, Estratégia de Armazenamento, etc.) devem ser passados via construtor da classe ClSearchManager. Uma vez que o método `realizarPesquisa()` é invocado, esses parâmetros são utilizados para a geração dos agentes e para o controle de sua execução. Dessa forma, não é

necessário que haja um contato direto com os agentes de software por parte dos instanciadores da aplicação, tudo fica escondido atrás da implementação dessa classe. Em grande parte, isto é devido à adoção de uma estratégia clara de separação entre a estrutura interna dos agentes e o sistema multi-agentes que compõe o *framework*, conforme proposto em [50, 51].

A Figura 9 mostra o diagrama de classe da camada de gerência do sistema multi-agentes.

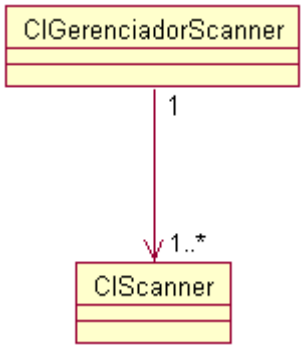

Figura 9: Camada de gerência do sistema multi-agentes