8
Instanciando o *Framework*

8.1
Classes a serem instanciadas

Como é ilustrado na Figura 1, o *framework* possui seis subsistemas, sendo que apenas o subsistema de gerência não possui nenhum *hot spot*. Não são muitas as dependências entre os subsistemas, logo é possível instanciar o *framework* de forma relativamente simples conceitualmente falando – ou seja, a dificuldade está na complexidade das classes instanciadas, e não no relacionamento entre elas. Ainda assim, existe uma ordem recomendada para instanciação, que procura guiar o instanciador a conseguir justamente o principal objetivo do *framework*: separar as questões referentes à busca e seleção dos documentos (plataforma) das referentes ao algoritmo de classificação utilizado, numa aplicação direta de *separation of concerns*. A dependência existente entre os subsistemas também é a mostrada na Figura 1.

É recomendado que se faça a instanciação em duas etapas:

- Inicialmente, deve-se tratar as questões de plataforma, representadas pelo subsistema Processador de Documentos, seguido do Subsistema de Memória, do Armazenador de Documentos a Visitar e, por último, do Gerador de Relatórios;
- Posteriormente, deve-se tratar as questões de classificação, representadas pelo Subsistema de Pesquisa.

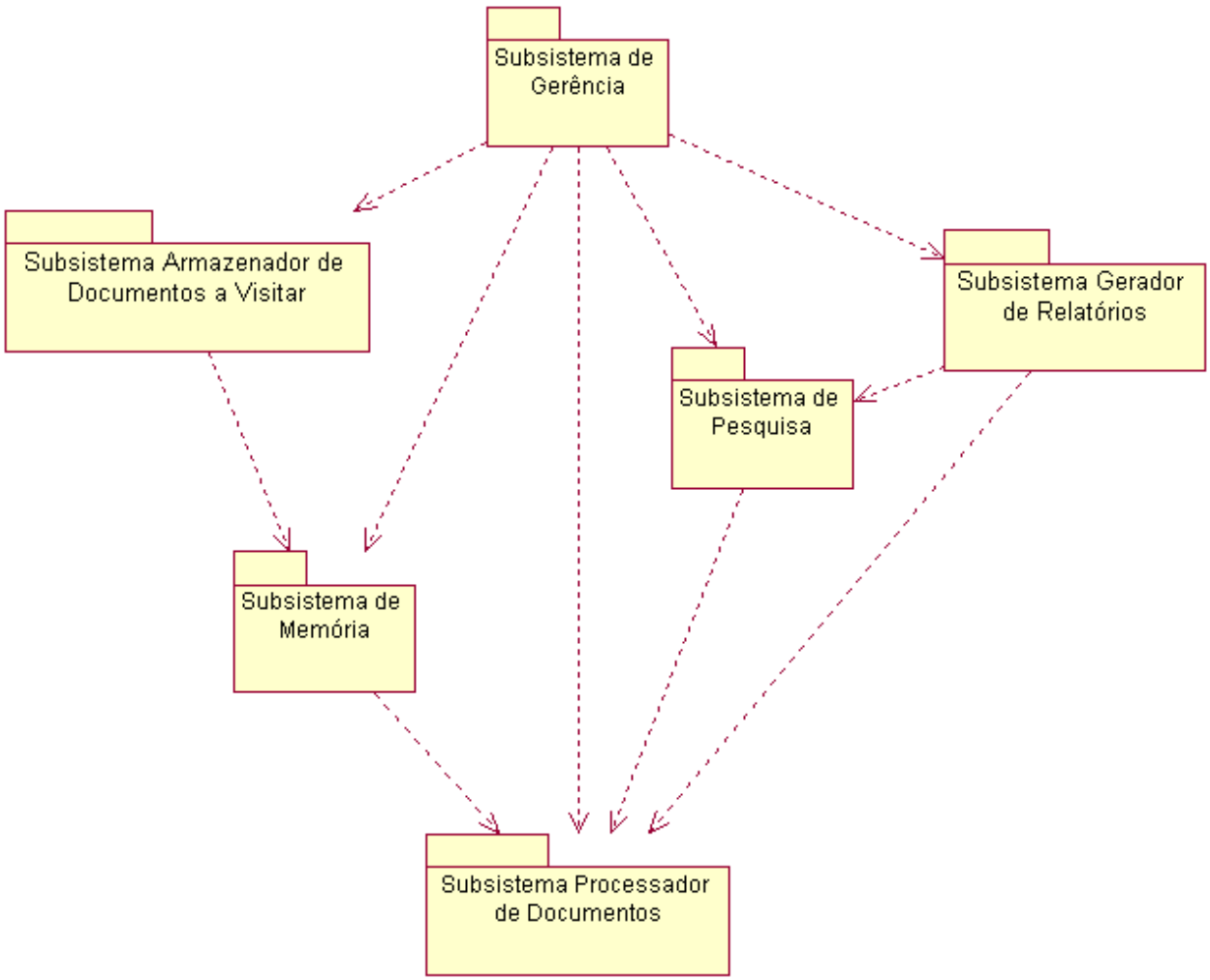

Figura 1: Subsistemas e dependências.

É importante que as questões abordadas pela segunda parte da instanciação sejam resolvidas por uma pessoa que entenda do domínio dos documentos que serão processados, a fim de que esta desenvolva um algoritmo específico e otimizado. Além disso, uma vez que as questões de plataforma já foram resolvidas na primeira parte da instanciação, é possível testar diversas formas (algoritmos) de classificação, possibilitando a escolha de um que melhor se enquadre nas necessidades da organização.

Apesar de haver um ponto de contato entre o Subsistema Gerador de Relatórios e o Subsistema de Pesquisa, o principal ponto de contato que existe entre as duas partes da instanciação é o momento em que é instanciado o

Subsistema Processador de Documentos, pois é necessário definir quais serão os parâmetros disponibilizados através dos métodos das classes que ficarem responsáveis pelo processamento dos documentos (classes que implementam a interface IDocumento), parâmetros esses que serão utilizados pelo algoritmo de classificação. Existem três possíveis abordagens para solucionar esse problema:

- Fazer o processamento do documento ser o mais completo possível, mesmo correndo o risco de ter parâmetros implementados que não serão usados;
- Fazer o processamento do documento ser o mais enxuto possível, já sabendo que serão necessárias algumas alterações a medida em que a segunda parte do *framework* for instanciada;
- Fazer com que o especialista do domínio dos documentos participe, ainda que em pequena escala, da instanciação inicial, a fim de definir quais parâmetros do documento serão disponibilizados.

Ainda assim, as modificações que porventura sejam necessárias no Processador de Documentos ficarão restritas apenas ao relacionamento entre esse subsistema e o de Pesquisa.

8.1.1 Questões de Plataforma

Esta é a primeira parte do *framework* que deve ser instanciada. Sugere-se que a primeira classe instanciada seja a que trata diretamente do processamento dos documentos: a IDocumento. Cada tipo de documento processado deverá ter uma instância separada dessa classe para tratá-lo.

É necessário que se defina um grupo inicial de parâmetros dos documentos que serão disponibilizados para a realização da classificação, logo é altamente desejável que o especialista no domínio dos documentos esteja presente nessa parte da instanciação. Idealmente, esses parâmetros devem ser os mesmos para todos os tipos de documentos, mas caso isso não seja possível, ou mesmo não seja desejável (pois alguns documentos podem ter características que facilitam o seu

processamento que não convém que sejam descartadas em prol da uniformidade), há suporte para processamento de múltiplos tipos de documentos simultaneamente.

Uma vez definidos os parâmetros, é necessário codificar a classe que o encapsula (que implementa a interface IDocumento), e o esforço dessa etapa será proporcional à dificuldade de obtenção dos parâmetros desejados – por exemplo, determinar o título de um documento no formato HTML (que está presente na *tag* TITLE) é muito mais fácil do que determinar em um documento no formato texto. Nota-se que o grau de estruturação e organização do documento é o fator determinante no esforço investido nessa parte da instanciação – quanto mais estruturado um documento for, mais fácil será obter parâmetros relevantes para a sua classificação.

Outro ponto importante a ser considerado é a geração do código hash de cada documento, valor que pode ser usado pelo subsistema de memória permanente (conforme já analisado anteriormente). Caso a memória não-volátil esteja ativa e a estratégia utilize códigos hash, é importante que o instanciador tenha em mente que, idealmente, dois documentos nunca podem retornar o mesmo código hash, senão serão considerados o mesmo documento.

O próximo passo é instanciar a classe geradora de documentos: a IFabricaDocumentos. No caso dos documentos processados serem de apenas um tipo, haverá apenas uma fábrica responsável por gerá-los. Porém, caso sejam processados diversos tipos de documentos simultaneamente, são possíveis duas abordagens: ou cria-se uma fábrica capaz de instanciar todos os documentos ou então gera-se uma fábrica para cada documento. Porém, conforme já mencionado anteriormente, o *framework* está preparado para lidar com apenas uma fábrica de documentos, logo é necessário encontrar uma forma de fazer com que o *framework* enxergue apenas uma fábrica, apesar de haver mais de uma. Uma possível abordagem é através de uma estrutura similar a do *pattern Chain of Responsability* [52] que possibilita que diversos objetos tenham a oportunidade de realizar uma tarefa de forma encadeada – os objetos são encadeados como numa corrente, de tal forma que, ao receberem uma requisição, têm a chance de verificar se são capazes de atendê-la e, caso não sejam, podem repassar essa requisição

adiante, para outros que objetos tentem atendê-la. O diagrama de classes e de objetos de um *Chain of Responsability* pode ser visto na Figura 2.

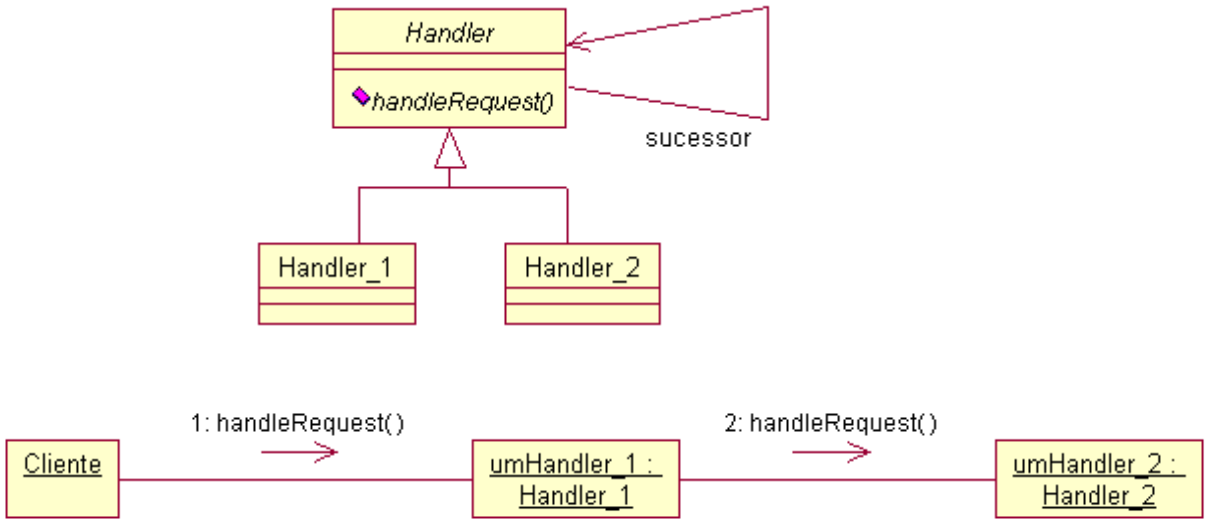

Figura 2: Diagrama de classes e objetos do *pattern Chain of Responsability*

O diagrama de classes para resolver o problema de múltiplos documentos é o presente na Figura 3, onde seria criada uma nova classe, denominada ClFabricaChain, que estaria diretamente relacionada com o *Handler* da estrutura do *pattern*: permitir apenas a formação da corrente (note que o método `processarDocumento` não está implementado). Cada uma das fábricas de documentos concretas seria instanciada a partir de ClFabricaChain, possibilitando a criação de uma estrutura de objetos idêntica à gerada pelo *Chain of Responsability*, muito embora o diagrama de classes seja ligeiramente diferente. Nessa estrutura (mostrada também na Figura 3), cada fábrica verifica se o documento modelado pelo ClDocumentoAVisitar pode ser instanciado por ela e, caso não possa, repassa à fábrica seguinte na corrente.

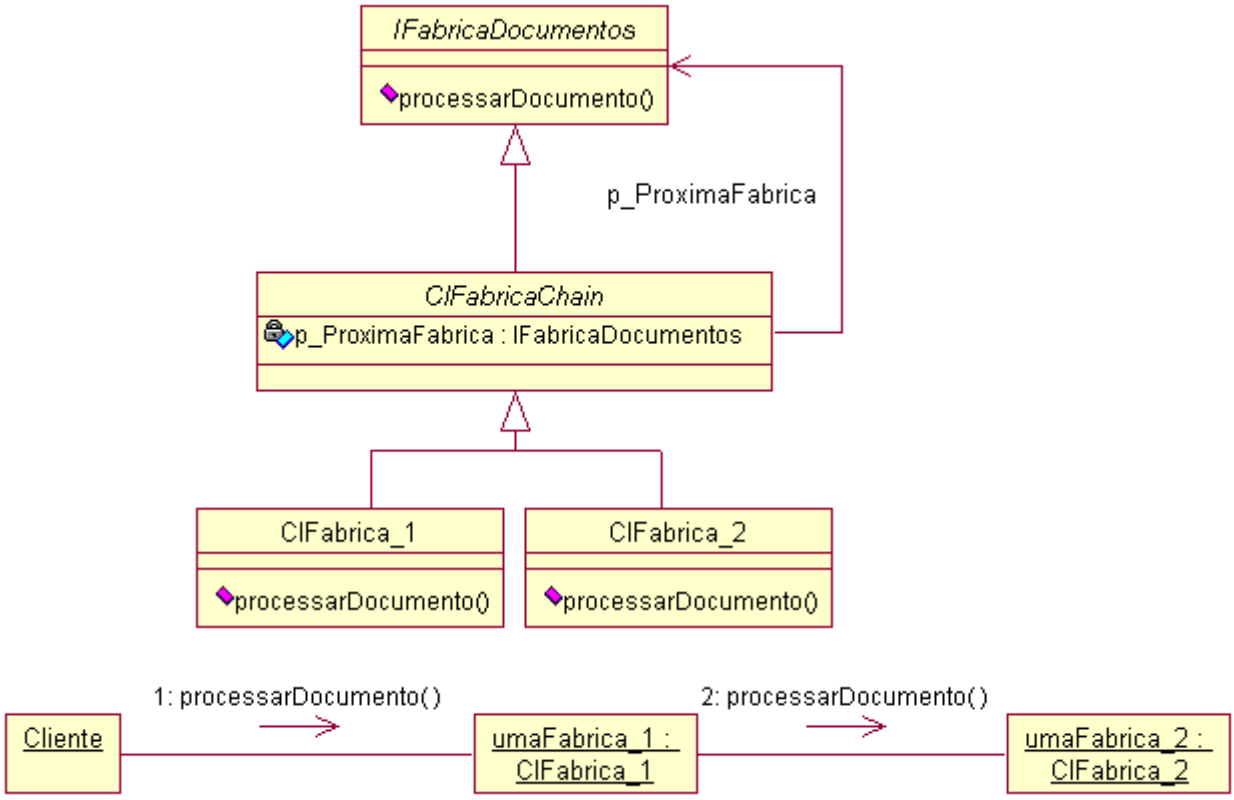

Figura 3: Diagrama de classes e objetos para o suporte ao processamento de múltiplos tipos de documentos

Em termos de Engenharia de Software, recomenda-se utilizar o *Chain of Responsability* porque ocorre uma separação melhor de tarefas – como cada fábrica é responsável por processar um tipo de documento, é mais fácil adicionar suporte ao processamento de um documento, bem como remover, se for o caso. Além disso, espera-se que o esforço empregado no desenvolvimento de classes separadas seja menor, uma vez que o código de cada uma será mais simples e o trabalho pode ser paralelizado.

Uma vez que as classes processadoras de documentos e as classes geradoras de documentos estejam instanciadas, o subsistema processador de documentos encontra-se completo. O próximo passo é instanciar a parte de memória não-volátil do subsistema de memória.

A instanciação da parte não-volátil do subsistema de memória é bastante simples: caso a memória permanente seja ativada, é necessário escrever uma

classe que implemente a interface IPagesKeeper, mas se for desativada, basta utilizar a classe ClDummyKeeper, já analisada anteriormente. Uma possível estratégia para a utilização da memória não-volátil é usar o esquema de hashs cuja infra-estrutura já está pronta. O armazenamento desses hashs pode ser feito num banco de dados, muito embora o simples armazenamento em um arquivo em formato texto ou binário já possa ser suficiente em aplicações que classificarão um número pequeno (ou controlado) de documentos.

O próximo passo é instanciar o subsistema gerenciador de documentos a visitar. Para isso, é necessário escrever uma classe que implemente a interface IArmazenamentoStrategy, ou então utilizar a classe ClSmartStorage, que é uma instanciação já fornecida pelo framework.

A última questão de plataforma é a instanciação do subsistema gerador de relatórios. O primeiro passo é escrever uma classe que especialize a classe ClRespostaPesquisa, que armazenará todos os parâmetros necessários para a geração do relatório. A natureza desses parâmetros já foi analisada anteriormente na discussão do subsistema de pesquisa. O segundo passo é escrever uma classe que implemente a interface ISender, que será responsável por armazenar todas as instâncias das subclasses de ClRespostaPesquisa e, uma vez finda a classificação, pode também gerar um relatório acerca dos resultados obtidos.

8.1.2 Questões de Classificação

Uma vez que as questões de plataforma já foram resolvidas, resta ao especialista no domínio dos documentos decidir qual será o algoritmo utilizado e implementá-lo através da instanciação do subsistema de pesquisa.

A classe mais importante (e também a única que ainda não foi instanciada até esse momento) para esse subsistema é a interface IPesquisa, que encapsula o algoritmo de classificação que será utilizado. Cada instância de uma classe que implemente IPesquisa representa a aplicação do algoritmo a uma única classe de documentos, ou seja, testa um documento acerca de sua pertinência a uma, e apenas uma, classe de documentos.

A estratégia de deixar cada instância tratar de apenas uma categoria torna possível o uso de algoritmos baseados em hierarquias de classes de uma forma direta: como cada classe é representada por uma instância, basta replicar a hierarquia existente no mundo real para uma estrutura equivalente entre os objetos da classe IPesquisa. Além disso, como não há limitação no número de pesquisas que o *framework* pode receber, também é possível utilizar classes de documentos que não tenham nenhum tipo de relacionamento entre si, ou seja, algoritmos que não são baseados em hierarquias.

A classe que implementar IPesquisa precisará escrever os três métodos definidos:

- `pesquisar` – recebe como parâmetro uma instância do documento a ser classificado, e retorna uma instância de ClRespostaPesquisa como resultado (na verdade, deve retornar uma instância da classe definida como a portadora dos dados para geração de relatórios que herda de ClRespostaPesquisa – subsistema gerador de relatórios). Não é necessário que este método retorne um objeto construído, há a possibilidade de retornar `null`. Caso isso ocorra, o *framework* entenderá que não há resposta conclusiva acerca do que foi pesquisado, e simplesmente ignorará o documento classificado.
- `ocorrerPesquisa` – Deve retornar um booleano indicando se o documento pertence ou não à classe modelada pela instância de IPesquisa. Caso haja algum grau de incerteza acerca do resultado, este método deve retornar "o melhor palpite" de acordo com o que foi processado.
- `toString` – Retorna uma representação em String do objeto.

8.2 Arquiteturas de instanciação

É possível utilizar diferentes arquiteturas durante a instanciação do *framework* de acordo com a natureza da aplicação a ser gerada. Essa flexibilidade

é necessária, pois não é possível determinar, *a priori*, quais serão os requisitos de performance e de garantias de execução de cada uma – determinadas aplicações podem ser mais críticas tanto em termos de tempo de execução quanto em termos de tolerância e recuperação de falhas de execução.

As questões de arquitetura aqui abordadas têm como principal foco propor formas para distribuir o processamento de maneira a conseguir instâncias escaláveis, bem como capazes de se recuperarem de falhas de execução.

O primeiro passo é separar essa questão em duas partes: a distribuição intra-processo (no nível de diferentes *threads* de execução) e a distribuição inter-processo, onde diferentes processos (provavelmente executando em diferentes máquinas) comporão o sistema final.

8.2.1 Distribuição Intra-processo

A distribuição intra-processo é representada, principalmente, pela arquitetura do sistema multi-agentes, que lida com distribuição de processamento dentro de um processo de classificação. Conforme mencionado, cada agente possui o seu próprio *thread* de execução, logo é possível ter um sistema multi-thread apenas alterando o número de agentes de software que executarão a classificação. Aqui cabe uma discussão importante sobre o real ganho com esse tipo de clusterização – conforme é citado em [55], podemos dividir *threads* e processos executando em uma mesma CPU em dois tipos:

- Intensivos em E/S (*I/O Bound*) – são *threads* e processos que passam a maior parte do seu tempo de execução realizando operações de entrada e saída, que são basicamente acessos a disco (leitura e escrita), acessos à rede (leitura e escrita em *sockets*) e acessos a periféricos em geral (impressora, *floppy drive*, etc.).
- Intensivos em uso de memória (*Memory Bound*) – são *threads* e processos que passam a maior parte do seu tempo de execução realizando operações em memória, que apenas consomem tempo de execução de CPU.

Podemos estender esse conceito criando uma terceira classe, composta pelos processos e *threads* que ficam alternando entre operações de ambas as naturezas.

Ora, é sabido que operações de entrada e saída são pelo menos uma ordem de grandeza mais lentas que operações realizadas em memória. Devido a isso, os sistemas operacionais mais modernos, como os da família UNIX e o Windows (que são os alvos desse trabalho por possuírem implementação de máquina virtual Java), são capazes de preemptar processos que solicitam operações de E/S, dando a oportunidade a outros processos de executar durante o tempo em que o processo solicitante fica inativo a espera do seu resultado. Muitos desses sistemas operacionais também são capazes de enxergar não só os processos, mas também seus *threads*, sendo capazes de realizar o controle nesse nível (Windows NT e Linux são exemplos). Nesses casos, fica claro que, em sistemas que possuem um único processador, não é possível obter ganhos expressivos de performance através de técnicas de distribuição intra-processo para processos ou *threads* que são intensivos em uso de memória. Porém, o mesmo não ocorre com os intensivos em E/S nem com os intermediários.

Logo, o primeiro passo antes de adotar uma política de criação de muitos agentes de software para compor o sistema, é verificar se o sistema operacional alvo é capaz de realizar preempção por *thread*.

Após a análise do sistema operacional alvo, antes de podermos estimar o quanto seria possível ganhar no *framework* instanciado, em termos de performance, através desse tipo de distribuição, é necessário estudar a natureza dos *threads* classificadores, ou seja, dos agentes de software. Um agente de software classificador fica constantemente executando os seguintes passos:

- Obtém um documento a ser classificado;
- Lê o documento;
- Extrai dele informações relevantes à classificação;
- Classifica-o, e gera um resultado de classificação.

É possível notar que, dependendo da forma como os *hot spots* forem instanciados, esses passos podem ter tanto operações de E/S como operações em

memória. Porém, os passos 1 e 2 têm tendência a ter mais operações de E/S, pois é o momento em que o documento será aberto e lido (de um disco rígido, de um disquete, de um *website*, de um banco de dados, etc.), ao passo que os demais passos lembram operações em memória, pois estão relacionadas ao processamento do documento. Além disso, a complexidade dos passos 3 e 4 varia de acordo com o algoritmo de classificação adotado (e complexidade é igual a maior tempo de execução de operações em memória), e o tempo de E/S gasto pelos passos 1 e 2 varia de acordo com o tamanho do documento. Assim, podemos considerar que *boa parte das aplicações instanciadas a partir do framework serão de natureza mista ou então intensivas em E/S*, ou seja, são capazes de ganhar com uma estratégia de distribuição intra-processo.

Outro ponto pertinente é que, como todos os agentes de software realizam a mesma tarefa, trata-se de um caso de clusterização e não de paralelização (conforme descrito em [56]). Porém, como cada agente de software reporta o resultado da classificação de um documento assim que ela acaba, pode ser interessante realizar paralelização na parte de geração de relatórios.

Por exemplo, tomemos um sistema onde esses resultados devem ser inseridos numa base de dados qualquer:

- Uma possível abordagem seria instanciar o *hot spot* de geração de relatórios de forma serial: cada agente insere seus resultados na base de dados assim que estes se tornam disponíveis. Nota-se que essa abordagem pode gerar problemas de performance, haja vista que pode ser difícil lidar com questões referentes ao compartilhamento de conexões com o banco de dados e também seria uma mistura de uma operação intensiva em memória (classificação) com uma operação intensiva em E/S (acesso a banco de dados).
- Outra abordagem seria criar *threads* armazenadores que seriam responsáveis unicamente por inserir no banco de dados os resultados das classificações de cada agente. Quando um agente de software terminasse uma classificação, ele colocaria o resultado em uma fila, compartilhada entre os diversos agentes e também entre os armazenadores. Um armazenador então seria alocado para remover essa resposta da fila e

armazená-la no banco de dados. Nota-se que é uma arquitetura de distribuição de processamento equivalente ao clássico problema dos produtores e consumidores [55], onde os agentes são os produtores e os armazenadores são os consumidores. Através dessa abordagem é possível obter ganhos de performance, concentrando-se uma operação intensiva em E/S em um outro *thread* diferente dos classificadores, e racionalizando-se o acesso ao banco de dados.

Outro exemplo da utilização de distribuição intra-processo é paralelizar a tarefa de processamento de documentos e a tarefa de classificação de documentos. Nessa arquitetura, parte do subsistema processador de documentos é separada de forma a permitir que os documentos sejam processados assim que forem encontrados, e não somente no momento em que são classificados. Dessa forma, os trabalhos de processamento e de classificação se dariam em paralelo. Para isso, são criados *Threads* cujo único objetivo é processar documentos. Cada um destes será doravante chamado de **processador**. Cada *Thread* de classificação (que se traduz num agente de software, conforme visto anteriormente) será chamado de **classificador**. A comunicação entre o processador e o classificador se dá através de duas listas: uma para conter os documentos a serem visitados e outra para conter os documentos já processados. A primeira será denominada **documentos a processar** e a segunda será denominada de **documentos processados**.

A lista de documentos a processar é alimentada pelos classificadores e consumida pelos processadores, ao passo que a lista de documentos processados é alimentada pelos processadores e consumida pelos classificadores. A seqüência temporal dos fatos ocorre da seguinte maneira:

- Em t=0, a lista de documentos a processar é alimentada com os documentos que inicialmente serão processados. Dispara-se então ambas as famílias de *Threads*: os processadores e os classificadores. Como não há documento na lista de documentos a classificar, os classificadores se tornam inativos. Já os processadores começarão imediatamente a consumir a lista de documentos processados e a alimentar a de documentos a classificar.

- Conforme a lista de documentos a classificar é preenchida, os classificadores são ativados. Conforme citado, durante a classificação de um documento há uma etapa em que se verifica se este pode indicar novos documentos a serem processados. Caso haja indicação, estas são inseridas na lista de documentos a processar.
- Esse processo continua até que ambas as listas fiquem vazias, pois como existe a possibilidade de um elemento de uma delas gerar elementos para a outra, nada se pode afirmar caso apenas uma fique vazia.

Nesse modelo de distribuição, há dois grupos *Threads* fazendo trabalhos distintos. Portanto, trata-se de um exemplo de paralelização e também de clusterização das tarefas de classificação e de processamento. É interessante notar que podemos encarar essa arquitetura como *criadora de um novo tipo de agente de software*, haja vista que os processadores podem ser encarados como agentes de software altamente reativos [38], cujo único objetivo é processar documentos.

Em termos de instanciação, o mapeamento das listas é o seguinte:

- A lista de documentos a processar deve ser modelada por uma classe que implemente a interface IArmazenamentoStrategy;
- A lista de documentos processados deve ser modelada por uma classe que implemente a interface IFabricaDocumentos. Deve ser criado um método para que o processador seja capaz de inserir um documento processado na lista. Como o classificador obtém documentos processados através do método `processarDocumento( ClDocumentoAVisitar DocumentoAVisitar )`, o método de inserção de um documento deve receber o `DocumentoAVisitar` correspondente para utilizá-lo como indexador (chave) para o documento correspondente (isso pode ser facilmente realizado de forma computacionalmente eficiente com uma tabela *hash*, que já está implementada na API de Java com o nome de *Hashmap*).
- O *Thread* de processamento deve ser modelado por uma classe que tenha acesso tanto à lista de documentos a processar quanto à lista de documentos processados.

A Figura 4 apresenta o diagrama de classes para essa arquitetura. Nela, o *Thread* processador é representado pela classe ClProcessador, que estende a classe Thread de Java. A fila de documentos a processar é modelada pela classe ClDocumentosAProcessar, e a de documentos processados pela ClDocumentosProcessados. As verdadeiras fábricas de documentos somente são vistas pelo processador.

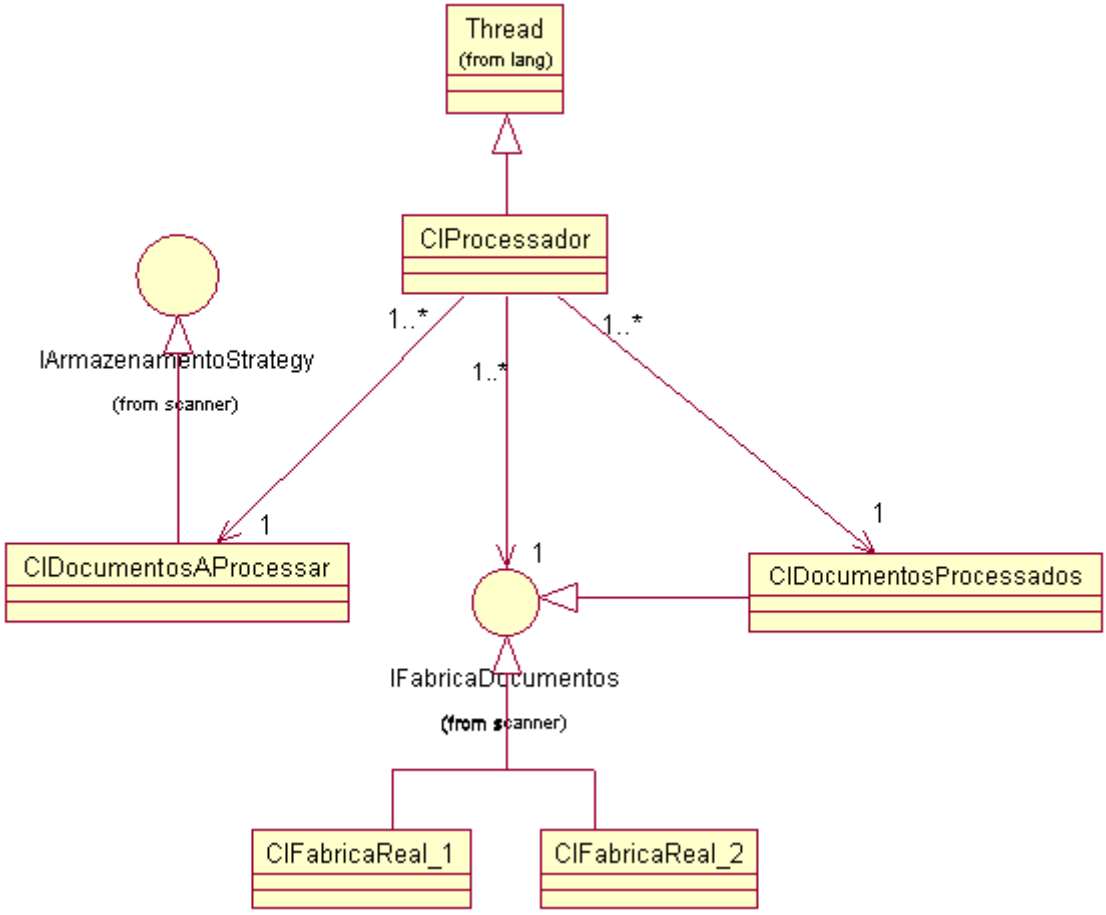

Figura 4: Diagrama de classes para a arquitetura dissociadora das tarefas de classificação e de processamento

8.2.2
Distribuição Inter-processo

Técnicas de distribuição inter-processo devem ser adotadas na instanciação do *framework* por dois motivos:

- Questões de escalabilidade e performance;

- Questões de confiabilidade na execução.

A fim de diminuir o tempo de desenvolvimento e aumentar a confiabilidade do sistema final, é recomendado o uso de algum *framework* para distribuição de processamento, como CORBA ou RMI (já aproveitando o fato de que o sistema foi escrito em Java).

Antes de discutirmos as arquiteturas de distribuição, é necessário definirmos quais as partes do sistema que podem ser distribuídas, e também que tipo de ganho é possível obter com a distribuição de cada uma. O diagrama de subsistemas componentes do *framework* encontra-se na Figura 1.

Dentre os subsistemas componentes, quatro se destacam por serem os grandes responsáveis pela utilização dos recursos totais utilizados:

- O subsistema de pesquisa e o subsistema processador de documentos são fortes candidatos a serem grandes utilizadores de tempo de CPU;
- O subsistema gerador de relatórios e o subsistema de memória são fortes candidatos a consumirem tanto recursos de CPU como recursos de E/S.

Não é possível afirmar com certeza a relação de recursos consumidos por cada um deles devido ao fato de todos possuírem alguns *hot spots* que, quando instanciados, podem ter impactos significativos nesse cálculo. Por exemplo, uma possível instanciação da parte de memória não-volátil do subsistema de memória seria armazenar os códigos *hash* em uma tabela de banco de dados relacional, a fim de poder consultá-los no futuro – uma estratégia que claramente pode gerar gargalos de E/S. A fim de evitar esse problema, poderia ser adotada uma estratégia de manter uma estrutura de *cache* intermediário, em memória, onde os *hashs* seriam carregados, em grupos, do banco de dados, segundo algum critério estabelecido. A fim de possibilitar ganhos ainda maiores de performance, essa estrutura poderia ser uma tabela *hash* implementada através de um vetor (que geraria, no caso da adoção de uma função de *hash* que distribuísse uniformemente os documentos ao longo da tabela, consultas em complexidade *O(1)*, ou seja, tempo constante) ou através de listas encadeadas ordenadas (que geraria, no caso da adoção de uma função de *hash* que distribuísse uniformemente os documentos ao longo da tabela, consultas em complexidade *O(log(N/M))*, onde N é o número

de *hashs* presentes em memória e M é o número de posições na tabela), uma árvore de busca binária ou ainda uma lista ordenada (ambas gerando consultas em *O(log(N))* onde N é o número de *hashs* presentes em memória). Nota-se que, nesse caso, o gargalo de E/S foi substituído por um gargalo de utilização de CPU. No caso extremo, onde não é necessária a memória não-volátil, esse subsistema não geraria nenhum tipo de gargalo de processamento.

Dessa forma, cabe ao instanciador do *framework* analisar, para o seu caso específico, onde estará o gargalo. Assim, o trabalho de apresentar arquiteturas totalmente fechadas, prontas para serem utilizadas, como numa receita de bolo, é inviável, pois cada uma se aplicaria a um caso muito específico. Uma alternativa para esse problema é apresentar arquiteturas mais simples, mas que possam ser combinadas, de forma a gerar arquiteturas mais complexas – tais arquiteturas serão doravante denominadas de *arquiteturas primitivas*.

Neste trabalho, propomos duas possíveis arquiteturas primitivas para distribuição (o que não significa que haja apenas duas). Porém, não foi possível realizar testes maiores para atestar as suas viabilidades tanto na parte de implementação quanto na parte de segurança que realmente resolvem o problema a que se propõem resolver.

As duas arquiteturas primitivas são as seguintes:

- **Arquitetura Espelho**: a arquitetura espelho tem por objetivo replicar a arquitetura do modelo de distribuição intra-processo;
- **Arquitetura Geradora**: a arquitetura geradora tem por objetivo separar a geração de relatórios da classificação dos documentos;

8.2.2.1 Arquitetura Espelho

Essa arquitetura é a mais simples de ser entendida, pois a necessidade de geração de novo código devido à distribuição é muito pequena. Apenas divide-se o grupo de documentos a serem classificados em subgrupos menores segundo alguma estratégia (por exemplo, a quantidade de documentos presentes em cada

grupo, ou o somatório do tamanho ocupado por todos eles – esses dois critérios devem servir como bons estimadores do tempo de processamento que cada unidade precisará), e cada um desses subgrupos é dado a um processo distinto. Ao final da classificação, cada processo terá abrangido um universo totalmente diferente de documentos.

Aqui cabem alguns comentários sobre como se comportam os demais subsistemas nessa arquitetura:

- Os subsistemas de gerência, de pesquisa, de documentos a visitar e de processamento de documentos, por não possuírem nenhum tipo de informação que deva ser armazenada de forma permanente, estão automaticamente preparados para essa arquitetura;
- O subsistema de geração de relatórios e a parte de memória não-volátil do subsistema de memória podem ter que sofrer algumas alterações, dependendo de como foram instanciados. Por exemplo, imagine que o subsistema de memória fosse implementado armazenando-se os códigos *hash* em uma base de dados relacional, conforme já mencionado anteriormente. Uma forma de implementar essa funcionalidade é criar uma tabela onde todos os processos inserem, atualizam e consultam os *hashs*. Porém, isso implicaria em perda de performance, uma vez que cada processo seria forçado a varrer todos os *hashs* dos demais processos que nada têm a ver com ele. Seria interessante que houvesse formas de evitar que um processo perdesse seu tempo com elementos do banco que não tivessem chance de pertencer ao grupo de documentos ao qual foi designado para classificar, por exemplo, separando os *hashs* em tabelas distintas de acordo com o grupo ao qual pertencem, ou então mantendo a estrutura de tabela única, mas adicionando informações que auxiliassem na sua indexação eficiente.

É possível perceber que esta arquitetura adota um modelo de clusterização, e não de paralelização [56], pois todos os processos executam exatamente a mesma tarefa. Outro ponto interessante é que essa arquitetura pode ser usada tanto para fins de ganho de performance como para fins de aumentar a tolerância a falhas do

sistema final, pois como os processos são independentes, a falha de um não acarretará problemas aos demais.

8.2.2.2 Arquitetura Geradora

Essa arquitetura tem por objetivo separar a geração de relatórios do resto do sistema. Para isso, é criado um processo separado, responsável apenas por tratar as respostas que cada agente de software gera durante o processo de classificação. Assim, haveria dois processos executando no sistema final:

- Um processo gerador, que fica com as responsabilidades de geração de relatórios (subsistema de geração de relatórios);
- Um processo classificador, contendo toda parte de classificação dos documentos (todos os subsistemas restantes);

Fica claro que, para ser utilizada essa arquitetura, é necessário utilizar algum mecanismo para comunicação inter-processo. Como o *framework* está escrito em Java, pode ser utilizado RMI, mas a utilização de CORBA ou DCOM também é possível. Todo o código de utilização remota deve ficar concentrado na classe que implementar a interface ISender, portanto, escondido do *framework*. Na Figura 5 é apresentado o diagrama de classes para essa arquitetura, onde existe a classe RemoteSender que é responsável por tratar da questão de distribuição.

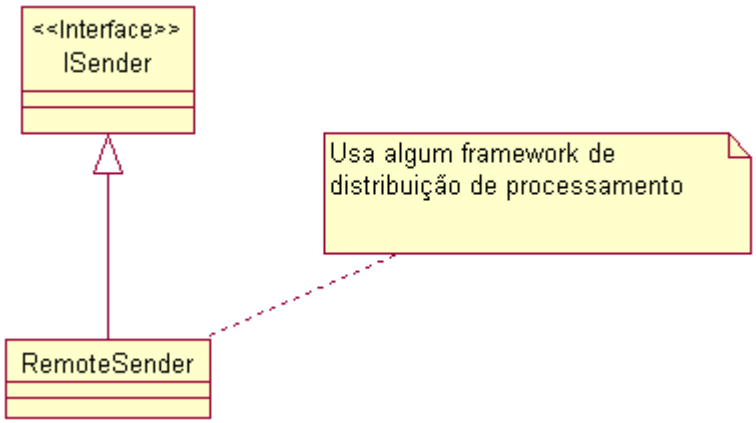

Figura 5: Diagrama de classes para a arquitetura geradora

Essa arquitetura dá margem a duas configurações de execução: podem ser utilizadas duas máquinas, uma para cada processo, ou então ambos podem executar numa mesma máquina (e esse processo é transparente caso o instanciador opte por utilizar CORBA, RMI ou DCOM).

A arquitetura geradora é especialmente útil em instâncias cujo subsistema gerador de respostas necessite de muito processamento em comparação com os demais subsistemas.