4
Implementação

O Capítulo 3 apresentou conceitualmente o sistema de recomendação para código de tratamento de exceção proposto nesta dissertação. Neste capítulo, serão apresentados detalhes de implementação do sistema. Considere, como exemplo, que o desenvolvedor está implementando o seguinte método, o qual lê um arquivo e retorna uma string contendo seu conteúdo:

```
public String readFileContent( File file ){
    StringBuffer buffer = new StringBuffer();

    FileReader fileReader =
           new FileReader(file); // thorws FileNotFoundException

    BufferedReader buffReader = new BufferedReader(fileReader);

    while(buffReader.ready()){ // throws IOException
           String line = buffReader.readLine();// throws IOException

           buffer.append(line);
    }

    String result = buffer.toString();

    return result;
}
```

Caso o desenvolvedor peça recomendações para o método acima, os seguintes fatos estruturais serão extraídos:

```
calls:java.io.BufferedReader.readLine
calls:java.io.BufferedReader.ready
calls:java.lang.StringBuffer.append
calls:java.lang.StringBuffer.toString
uses:boolean
uses:java.io.BufferedReader
uses:java.io.File
uses:java.io.FileReader
uses:java.lang.String
uses:java.lang.StringBuffer
handles:java.io.FileNotFoundExceptionException
handles:java.io.IOException
```

Com base nestes fatos estruturais, uma das recomendações realizadas será a seguinte:

```
private String getMergeResolveMessage(Repository mergeRepository) {
    File mergeMsg =
           new File(mergeRepository.getDirectory(),
                    Constants.MERGE_MSG);

    FileReader reader;

    try {
           reader = new FileReader(mergeMsg);

           BufferedReader br = new BufferedReader(reader);

           try {
                  StringBuilder message = new StringBuilder();

                  String s;

                  String newLine = newLine();

                  while ((s = br.readLine()) != null)
                         message.append(s).append(newLine);


                  return message.toString();

           } catch (IOException e) {
                  throw new IllegalStateException(e);
           } finally {
                  try {
                         br.close();
                  } catch (IOException e) {
                         throw new IllegalStateException(e);
                  }
           }
    } catch (FileNotFoundException e) {
           String message = NLS.bind(UIText.CouldNotFindMergeMsg,
                  Constants.MERGE_MSG);

           IStatus status = new CompanyStatus(
                  IStatus.ERROR,Activator.PLUGIN_ID,
                  CompanyStatus.CONFIGURATION_ERROR,
                  message,
                  exception);

           StatusManager.getManager().handle(
                  status,
                  StatusManager.LOG);

           return message;
    }
}
```

No exemplo de código recomendado acima, estão grifadas as informações relacionadas ao tratamento de exceções que são úteis ao desenvolvedor. Perceba no bloco `finally` do exemplo acima, que há uma chamada ao método `close()`. Esta informação é importante para que o desenvolvedor não se esqueça de desalocar os recursos previamente alocados, evitando assim problemas de desempenho em seu sistema. Nos blocos catch que tratam as exceções do tipo `IOException`, a exceção tratada é remapeada para o tipo não checado

`IllegalStateException`. Esta é uma possibilidade de indicar ao módulo cliente a ocorrência destas exceções. Já o bloco `catch` que trata a exceção `FileNotFoundException`, realiza *logging* de uma mensagem de erro. Para tanto, acessa um arquivo de mensagens através do método `NLS.bind`, cria uma instância de `IStatus` e passa esta instância para o gerenciador do framework Eclipse. O acesso a arquivos de mensagens através do método `NLS.bind` e o *logging* através do `StatusManager` são, inclusive, diretrizes de uso definidas pelo *framework* Eclipse. Percebe que com a recomendação realizada pelo sistema de recomendação, o desenvolvedor dispõe de exemplos concretos de como é possível tratar suas exceções de maneira mais adequada. Ainda que a realização de *logging* seja uma tarefa bastante simples, perceba como esta tarefa pode tornar-se bastante complexa quando se trabalha com desenvolvimento de aplicações baseadas em *frameworks*. Sem um suporte adequado, o desenvolvedor teria bastante dificuldade em encontrar este tipo de informação.

No restante deste capítulo serão discutidos os detalhes de implementação do sistema de recomendação proposto. Na Seção 4.1 apresenta-se a arquitetura de software do sistema de recomendação, apresentando em detalhes os módulos principais do sistema. A Seção 4.1.1 apresenta o Módulo Extrator de Informações, a Seção 4.1.2 descreve o Módulo Gerenciador de Repositório e a Seção 4.1.3 detalha o Módulo Recomendador.

4.1. Arquitetura de software

A implementação do sistema foi realizada usando a linguagem de programação Java. O sistema de recomendações pode ser dividido em três grandes módulos básicos: o Módulo Extrator de Informações, o Módulo Gerenciador de Repositório e o Módulo Recomendador. A Tabela 2 mostra quais as responsabilidades de cada um destes módulos.

Tabela 2 Módulos e responsabilidades

Módulo	Responsabilidade
Extrator de Informações	- Extrair fragmentos de código - Extrair fatos estruturais - Detectar fragmentos de código implementando tratadores ineficazes
Gerenciador de Repositório	- Armazenar fragmentos de código - Indexar fragmentos de código
Recomendador	- Construir consultas - Executar consultas - Pontuar candidatos - Ordenar candidatos

O Módulo Extrator de Informações tem como responsabilidades: (i) extrair do código das aplicações-fonte fragmentos de código a serem armazenados no repositório de exemplos e (ii) extrair fatos estruturais de um determinado fragmento de código. Além disso, o Módulo Extrator deve detectar fragmentos de código que implementem tratadores ineficazes, para evitar que tais fragmentos de código sejam inseridos no repositório de exemplos. O Módulo Gerenciador de Repositório tem como responsabilidades armazenar e indexar os fragmentos de código do repositório de exemplos. O Módulo Recomendador tem como responsabilidades construir e executar as consultas ao repositório, pontuar e ordenar os candidatos selecionados pelas consultas. Os três módulos principais colaboram entre si a fim de prover os serviços do sistema de recomendações para código de tratamento de exceções. O diagrama da Figura 10 mostra as colaborações entre os módulos.

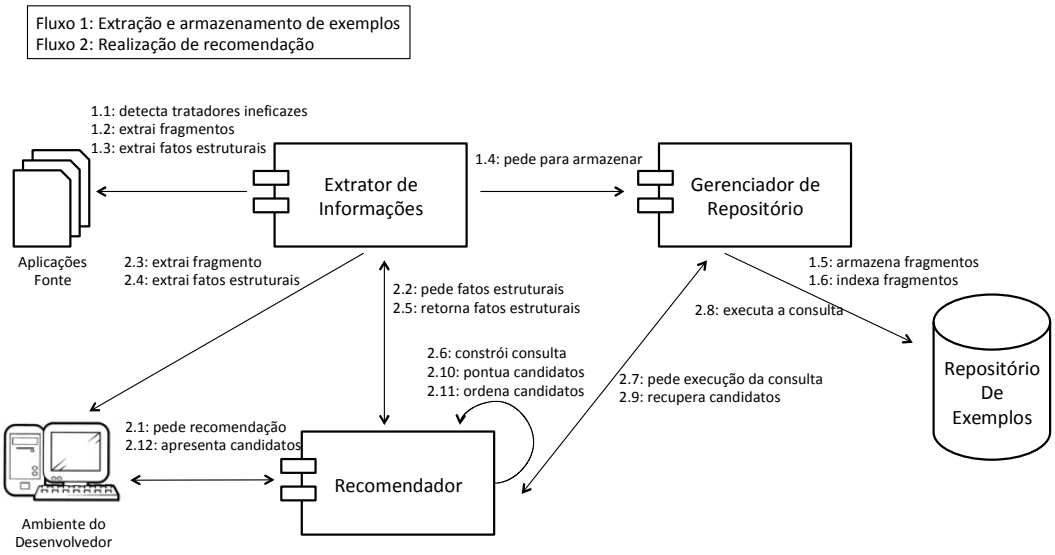

Figura 10 Colaboração entre módulos

No diagrama da Figura 10 acima são descritos dois fluxos de interação entre os módulos. O Fluxo 1 (definido pelas setas com rótulos prefixados com 1) descreve o fluxo de interação responsável por extrair os fragmentos de código das aplicações fonte e armazená-los no repositório de exemplos. Já o Fluxo 2 (definido pelas setas com rótulos prefixados com 2) descreve o fluxo de interação responsável por realizar as recomendações.

O Fluxo 1 tem início com o Módulo Extrator de Informações analisando o código fonte das aplicações-fonte. Todas as declarações de método contidas no código fonte das aplicações-fonte são analisadas, considerando-se apenas aquelas que possuem ao menos um bloco `catch`. Caso uma declaração de método não contenha um bloco `catch`, ou contenha um que esteja implementando um tratador ineficaz (MCCUNE, 2006; CHEN, 2008), esta será ignorada pela análise. Caso contrário, o fragmento de código correspondente a esta declaração de método será extraída, terá seus fatos estruturais extraídos e será indicada para inserção no repositório de exemplo. Após todas as declarações de método de um arquivo terem sido analisadas, o Módulo Extrator de Informações faz uma requisição ao Módulo Gerenciador de Repositório pedindo o armazenamento dos fragmentos de código extraídos, passando os fragmentos de código e os respectivos fatos estruturais como parâmetro. O Módulo Gerenciador de Repositório cria um novo registro correspondente a cada fragmento de código passado pelo Módulo Extrator de Informações e usa os fatos estruturais deste

fragmento de código para indexar o novo registro do repositório. Este fluxo de interação é repetido para cada arquivo fonte Java contido nas aplicações-fonte.

O Fluxo 2 tem início com o desenvolvedor realizando um pedido de recomendação ao Módulo Recomendador através do seu ambiente de desenvolvimento. Ao receber tal requisição, o Módulo Recomendador requisita ao Módulo Extrator de Informações os fatos estruturais do código em que o desenvolvedor está trabalhando. O Módulo Extrator de Informações extrai o fragmento em que o desenvolvedor está trabalhando, extrai os fatos estruturais deste fragmento e os retorna para o Módulo Recomendador. O Módulo Recomendador, de posse dos fatos estruturais do código em que o desenvolvedor está trabalhando, constrói uma consulta e a executa no repositório de exemplos. A execução é realizada através do Módulo Gerenciador de Repositório, o qual recupera os candidatos que satisfazem a consulta. Em seguida, para cada candidato, o Módulo Recomendador calcula a sua nota de relevância em relação à consulta usada, conforme definido na Seção 3.4.2.2. Os candidatos são então ordenados de acordo com estas notas. Os N primeiros candidatos na lista ordenada por relevância, em que N é um valor ajustado pelo desenvolvedor-usuário, são então retornados ao desenvolvedor-usuário. Nas Seções seguintes a implementação de cada um dos módulos será apresentado em mais detalhes.

4.1.1. Módulo Extrator de Informações

Como definido na Tabela 2, o Módulo Extrator de Informações possui as seguintes responsabilidades:

- Extrair fragmentos de código
- Extrair fatos estruturais
- Detectar fragmentos de código implementando tratadores ineficazes

Todas estas responsabilidades são realizadas através de um analisador sintático para a linguagem Java implementado especificamente para realizá-las. O analisador sintático recebe como entrada um arquivo contendo código fonte Java e transforma este código em uma árvore sintática. Uma árvore sintática nada mais é do que a representação sintática em uma estrutura de dados árvore do código fonte de acordo com a gramática formal que define a linguagem Java. A Figura 11 mostra a arquitetura do Módulo Extrator de Informações.

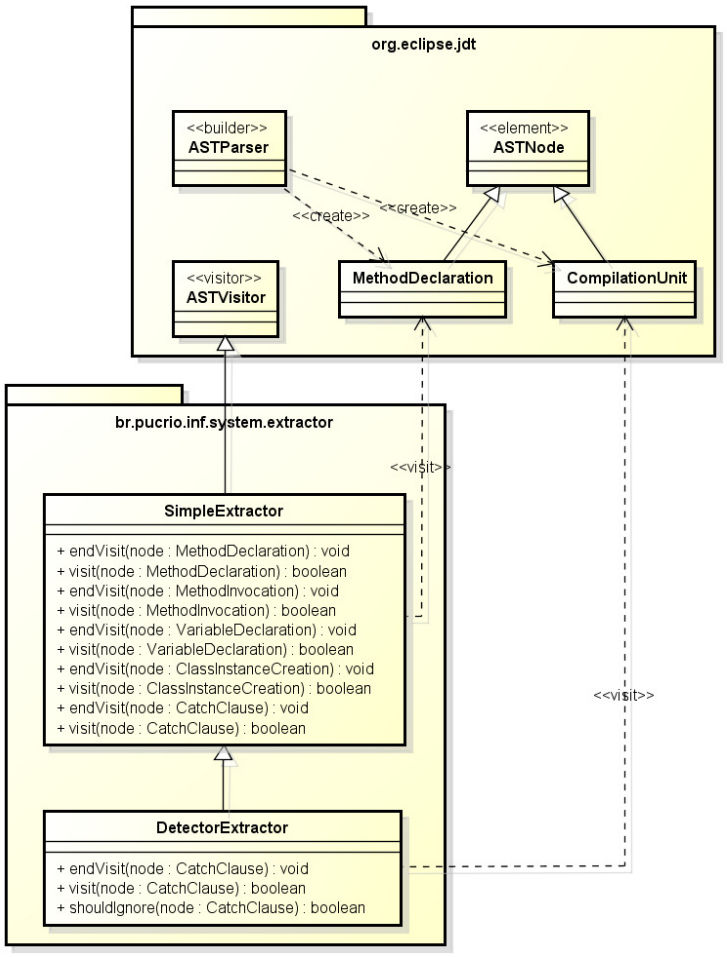

Figura 11 Arquitetura do Módulo de Extração de Informações

Como mostra a Figura 11, o Módulo de Extração de Informações foi implementado usando o *framework Eclipse Java Development Tool* – JDT. O JDT provê uma infra-estrutura básica para a implementação de analisadores sintáticos para a linguagem Java. A classe `ASTParser` implementa o padrão de projetos *Builder* (GAMMA et al, 1995) para a construção de árvores sintáticas. As árvores sintáticas são representadas como classes que herdam da classe `ASTNode`. A classe `ASTParser` constrói árvores sintáticas a partir de cadeias de caracteres representando fragmentos de código fonte Java. Os fragmentos de código podem ser definidos em diferentes níveis de granularidade, desde um arquivo Java completo, até uma única instrução Java. O Módulo de Extração de Informações trabalha com fragmentos de código em duas granularidades: tanto com fragmentos de código no nível de um arquivo Java completo, quanto no nível da declaração de método. Os fragmentos de código no nível de um arquivo Java completo são

analisados durante o processo de extração de exemplos (Fluxo 1, Seção 4.2). Suas árvores sintáticas são representadas por instâncias da classe `CompilationUnit`. Já os fragmentos de código no nível da declaração de método são analisados durante o processo de recomendação (Fluxo 2, Seção 4.2), quando se extraem os fatos estruturais do código em que o desenvolvedor está trabalhando. Suas árvores sintáticas são representadas por instâncias da classe `MethodDeclaration`.

As árvores sintáticas criadas pela classe `ASTParser` são implementadas como elementos visitáveis do padrão de projetos *Visitor* (GAMMA et al, 1995). Para visitar estes elementos, é necessário definir uma extensão à classe base `ASTVisitor`. O Módulo de Extração de Informações define duas estratégias de visitação às árvores sintáticas: `DetectorExtractor` e `SimpleExtractor`.

A estratégia implementada pela classe `SimpleExtractor` refere-se à estratégia de extração usada durante o processo de recomendação. Neste processo, é necessário apenas extrair os fatos estruturais de uma determinada declaração de método. A classe `SimpleExtractor` é usada para visitar árvores sintáticas representadas pela classe `MethodDeclaration`. Já a estratégia implementada pela classe `DetectorExtractor` é usada durante o processo de extração de exemplos. Neste processo, é necessário inicialmente extrair todos os fragmentos de código no nível da declaração de métodos de um determinado arquivo. Por isso, a classe `DetectorExtractor` é usada para visitar árvores sintáticas instâncias de `CompilationUnit`. Além de extrair os exemplos, também é necessário ignorar os exemplos que implementam tratadores de exceções ineficazes. Para tanto, a classe `DetectorExtractor` verifica no momento em que se visita uma declaração de método que contém bloco `catch`, se este bloco implementa um tratador ineficaz. Caso implemente, a visita à essa declaração de método é interrompida, seguindo para a declaração de método seguinte. Caso contrário, a extração de fatos ocorre normalmente.

4.1.2.
Módulo Gerenciador de Repositório

Como definido na Tabela 2, o Módulo Gerenciador de Repositório possui as seguintes responsabilidades:

- Armazenar fragmentos de código
- Indexar fragmentos de código

A Figura 12 mostra a arquitetura do Módulo de Gerenciamento de Repositório.

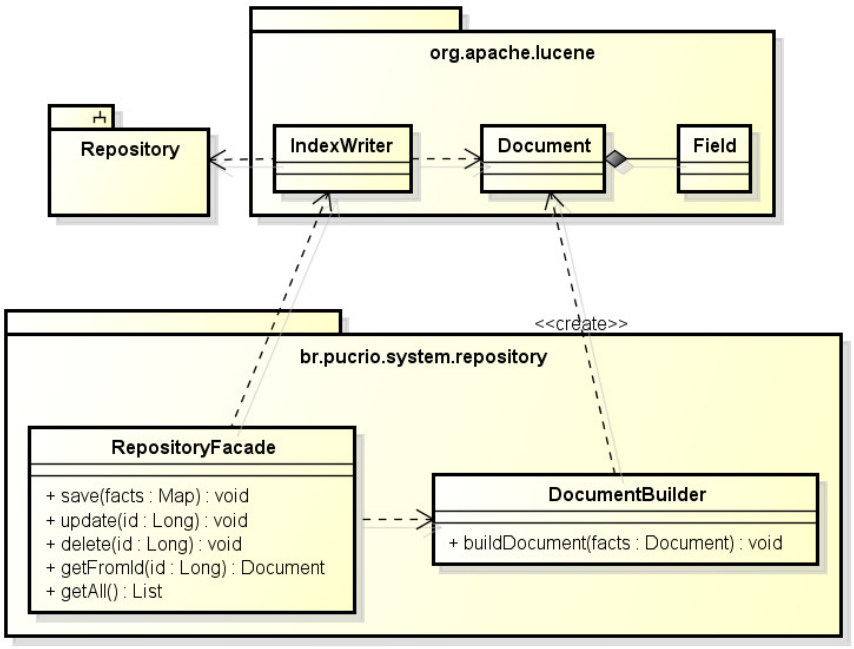

Figura 12 Arquitetura do Módulo de Gerenciamento de Repositório

Como mostrado na Figura 12, para a implementação do Módulo de Gerenciamento de Repositório foi usado o *framework* Apache Lucene, o qual provê um conjunto de funcionalidades para criação de ferramentas de buscas baseadas em texto. Especificamente, para a implementação do Módulo de Gerenciamento de Repositório foi usado o suporte à criação e manipulação de índice de dados.

O repositório de exemplos mantido pelo Módulo de Gerenciamento de Repositório é na verdade um grande índice de dados criado com o Lucene. Os índices de dados criados e manipulados pelo Lucene são implementados como uma estrutura de dados *Índice Reverso*. Em poucas palavras, um índice reverso associa um conteúdo (como uma palavra ou número, por exemplo) a um conjunto

de documentos que de alguma forma estão relacionados a este conteúdo. No contexto do sistema de recomendação para código de tratamento de exceções, o repositório de exemplos é, na prática, um índice que associa fatos estruturais a conjuntos de métodos que possuem aquele fato. Desta forma, dado um fato estrutural é possível identificar quais métodos no repositório o possuem.

As funcionalidades de criação e manipulação dos índices são providas pela classe `IndexWriter`. O Módulo de Gerenciamento de Repositório acessa estas funcionalidades através da classe `RepositoryFacade`. A classe `RepositoryFacade` recebe um conjunto de fatos estruturais relacionados a um determinado método e insere estas informações no repositório. Os fatos estruturais que são extraídos pelo Módulo de Extração de Informação são representados internamente como tabelas de dispersão que associam um campo (`Uses`, `Calls` ou `Handles`) ao seu respectivo conjunto de valores. Já os índices criados pelo Lucene representam as unidades armazenadas como instâncias da classe `Document`. Uma instância `Document` é composta por um conjunto de instâncias da classe `Field`. A classe `Field`, por sua vez, define um campo a ser armazenado no índice. Fazendo um paralelo com modelos relacionais de bases de dados, um `Document` é equivalente a uma tupla (linha em uma coluna) e um `Field` é um par `(coluna, valor)`. Para converter a tabela de dispersão que representa os fatos estruturais de um fragmento de código em um `Document`, implementou-se a classe `DocumentBuilder`. Para cada conjunto de fatos extraído de candidato, a classe `DocumentBuilder` constrói uma instância de `Document` e representa cada fato estrutural como uma instância da classe `Field`. Além das instâncias da classe `Field` relacionadas aos fatos estruturais, também se constrói uma instância de `Field` para armazenar a cadeia de caracteres representando o código fonte do exemplo armazenado.

Desta forma, quando uma solicitação de inserção de candidato ao repositório é recebida pela classe `RepositoryFacade`, primeiramente converte-se o conjunto de fatos estruturais para uma instância de `Document`. Em seguida, esta instância de `Document` é passada para a classe `IndexWriter`, que é quem de fato persiste o candidato no repositório de exemplos.

4.1.3.
Módulo Recomendador

Como definido na Tabela 2, o Módulo Recomendador possui as seguintes responsabilidades:

- Construir consultas
- Executar consultas
- Pontuar candidatos
- Ordenar candidatos

A Figura 13 mostra a arquitetura do Módulo Recomendador.

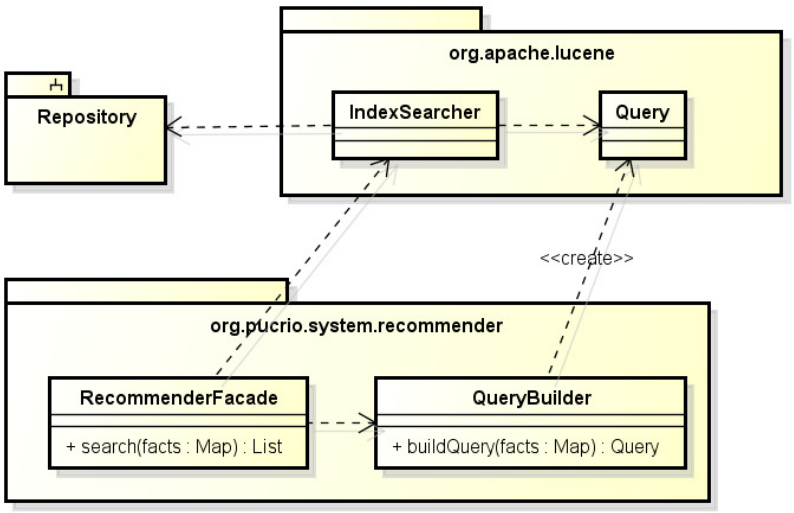

Figura 13 Arquitetura do Módulo Recomendador

O Módulo Recomendador mostrado na Figura 13 também foi implementado com base no *framework* Apache Lucene. Especificamente, foram usadas as funcionalidades de suporte a buscas em índices de dados. As funcionalidades de busca em índices de dados são providas pela classe `IndexSearcher`. O Módulo Recomendador acessa estas funcionalidades através da classe `RecommenderFacade`. A classe `RecommenderFacade` recebe um conjunto de fatos estruturais extraídos do fragmento de código em que o desenvolvedor está trabalhando para realizar uma consulta ao repositório em busca de candidatos a recomendação. Entretanto, a classe `IndexSearcher` realiza consultas a índices criados pelo Lucene com base em consultas em uma linguagem própria definida pelo *framework*. Estas consultas são representadas como instâncias da classe

`Query`. A fim de se construir instâncias da classe `Query` com base em um conjunto de fatos estruturais implementou-se a classe `QueryBuilder`.

A linguagem de consultas do Lucene é baseada em um sistema booleano não estrito. Em outras palavras, as consultas são descritas através de termos e estes termos podem ser combinados em expressões através do uso de operadores, como em um sistema booleano convencional, mas os operadores não são os operadores convencionais (E, OU e NEGAÇÃO). A linguagem de consulta do Apache Lucene define os seguintes operadores:

- MUST
- MUST_NOT
- SHOULD

Visto que as consultas definidas na Seção 3.4.2.1 são definidas como expressões booleanas disjuntivas, para realizar a tradução das consultas do sistema de recomendações para a linguagem do Lucene foi necessário apenas definir um mapeamento para o operador OU. Portanto, o operador OU foi mapeado para o operador SHOULD, o qual possui comportamento mais similar com o especificado na Seção 3.4.2.1. O operador SHOULD define que se espera que um candidato satisfaça um termo da consulta, mas não há obrigação para que este termo seja satisfeito. Por exemplo, se uma consulta "SHOULD(Fato1) SHOULD(Fato2)" encontra um candidato que contém o Fato1, mas não contém o Fato2, ele será selecionado mesmo assim, ainda que não satisfaça o termo SHOULD(Fato2). Se por um acaso nenhum candidato possuir nem o fato Fato1 ne, o Fato2, então a consulta retornará vazio.

A classe `RecommenderFacade` recebe de `QueryBuilder` uma instância da classe `Query` e a repassa para a classe `IndexSearcher`. A classe `IndexSearcher` é quem de fato executa a consulta no repositório, retornando os exemplos que satisfazem a uma determinada consulta. De posse dos exemplos retornados, a classe `RecommenderFacade` realiza a pontuação de cada candidato conforme definido pela função de pontuação da Seção 3.4.2.2. Após a pontuação, os candidatos são ordenados em ordem decrescente de nota recebida. Por fim, os N primeiros candidatos ordenados são recomendados.