3
Sistema de recomendação para código de tratamento de exceções

Neste capítulo, é apresentado o sistema proposto para recomendação de código de tratamento de exceções. Optou-se por implementar um sistema de recomendação para código de tratamento de exceção em resposta à limitação do estado arte (Seção 2.1.1). Em especial, a limitação das ferramentas e técnicas propostas para auxiliar desenvolvedores de software a lidar com tratamento de exceções em sistemas de software. Tais ferramentas e técnicas propostas seguem todas um mesmo paradigma, em que primeiro se detecta a má qualidade do código de tratamento de exceções para em seguida melhorias serem realizadas. Mas o que os estudos preliminares realizados nesta dissertação (Seção 2.3) sugerem é que esta abordagem não é muito eficaz. Em cenários de evolução de projetos de software o momento em que estas melhorias são realizadas geralmente ocorrem em momentos muito tardios (BARBOSA e GARCIA, 2010). Nestes cenários de melhoria tardia, o mau projeto do tratamento de exceções de um sistema pode até comprometer o projeto do sistema como um todo. Há ainda cenários de evolução piores, em que tais melhorias nunca ocorrem (BARBOSA e GARCIA, 2010).

Um sistema de recomendação capaz de prover exemplos de código que implementa tratamento de exceções é uma tentativa de quebrar o paradigma seguido pelas ferramentas já propostas. Espera-se que o provimento de exemplos de código seja capaz de auxiliar os desenvolvedores a implementar código de tratamento de exceções mais adequado desde o princípio. O aprendizado através de exemplos é, inclusive, apontado por alguns desenvolvedores como uma forma eficaz para aprender sobre novas formas de realizar tratamento de exceções (SHAH *et al.*, 2008a).

O restante deste capítulo está estruturado da seguinte maneira. Na Seção 3.1 são discutidas as restrições e decisões que guiaram o projeto do sistema de recomendação proposto. Na Seção 3.2 são apresentados conceitos gerais e

nomenclaturas utilizados no restante deste capítulo, bem como uma visão geral do funcionamento do sistema de recomendação. Na Seção 3.3 é apresentado o mecanismo de coleta de informações do sistema de recomendação e na Seção 3.4 é apresentado o mecanismo de recomendação do sistema de recomendação.

3.1. Restrições e decisões de projeto

A construção do sistema de recomendação para código de tratamento de exceções não foi uma tarefa trivial. Nesta seção estão descritas algumas das dificuldades e limitações que guiaram o projeto do sistema de recomendação para código de tratamento de exceções.

A primeira restrição tomada ao desenvolver o sistema de recomendação foi limitar as recomendações a exemplos de código implementados em Java. Como discutido na Seção 2.1, os elementos de tratamento de exceções são implementados de maneiras distintas em cada linguagem de programação, ainda que sintaticamente sejam parecidos (GARCIA *et al.*, 2001). Desta forma, não faz sentido prover exemplos implementados em uma linguagem de programação para desenvolvedores que estão usando outra linguagem. A opção por Java deveu-se pelo fato de que esta linguagem é atualmente uma linguagem orientada a objetos bastante utilizada, tanto academicamente quanto comercialmente. Desta forma, seria mais fácil encontrar aplicações-fonte, que serviriam de base para a extração dos exemplos a serem recomendados.

Uma preocupação tomada durante o projeto e o desenvolvimento do sistema de recomendação para código de tratamento de exceções foi em relação à adequação das recomendações. Para facilitar o aprendizado dos desenvolvedores, seria importante que os exemplos de código recomendados implementassem tratamento de exceções em tarefas similares à tarefa que o desenvolvedor está implementando. Para tanto, foram definidas informações estruturais que tentam caracterizar o contexto de um fragmento de código. As heurísticas de busca do sistema de recomendação (Seção 3.4.1) usam um conjunto dessas informações estruturais para realizar suas buscas. Tais informações estruturais são extraídas do código em que o desenvolvedor está trabalhando para que as heurísticas busquem por exemplos de código que possuam informações estruturais similares. Desta

forma, tenta-se maximizar a chance de encontrar exemplos com tratamento de exceções em contexto semelhante ao contexto do código que o desenvolvedor está trabalhando. Mais detalhes a respeito das heurísticas de busca serão discutidos na Seção 3.3.

Uma dificuldade a ser contornada durante o desenvolvimento do sistema de recomendação para código de tratamento de exceções foi encontrar os exemplos a serem recomendados aos desenvolvedores. Esta dificuldade foi parcialmente resolvida em vista o grande número de aplicações de código aberto disponíveis em repositórios públicos e que poderiam servir de fontes de exemplos. No entanto, não foram quaisquer aplicações que serviram como fonte para a extração de exemplos. Era necessário que as aplicações usadas apresentassem código de tratamento de exceções bem estruturado, ou pelo menos de qualidade minimamente aceitável, para que os exemplos de código extraídos fossem de boa qualidade . Assume-se que quanto melhor as informações disponíveis, melhor serão as recomendações realizadas. Mas como mostrado na Seção 2.1, estudos passados mostram que aplicações em geral, mesmo aquelas usadas na indústria, apresentam código de tratamento de exceções mal estruturado. Desta forma, foi necessário realizar uma pesquisa e um processo de seleção de projetos que tivessem código de tratamento minimamente estruturado. Além desta pesquisa e do processo de seleção, o processo de extração de exemplos ignorou aqueles que implementam os maus tratadores definidos em (MCCUNE, 2006) e em (CHEN *et al.*, 2009). Com isto, tentou-se ao menos evitar a extração de maus tratadores. Mais detalhes a respeito do mecanismo de coleta de informações são discutidos na Seção 3.3.

3.2. Visão geral

Este trabalho tem como objetivo desenvolver um sistema de recomendação baseado no conteúdo que seja capaz de prover a um desenvolvedor recomendações que o auxiliem na tarefa de tratar exceções no contexto do desenvolvimento de aplicações de software. As recomendações são realizadas através do provimento de exemplos concretos de código previamente

implementados. Os desenvolvedores usuários do sistema de recomendação proposto podem se beneficiar das recomendações nas seguintes situações:

- Enquanto implementando um novo tratador de exceção;
- Enquanto melhorando algum tratador de exceção previamente implementado.

Ressalta-se que não é objetivo deste trabalho prover exemplos de código para reuso *ipsis litteris*. O objetivo é prover exemplos de código em uma lista ranqueada por ordem de relevância que auxilie o desenvolvedor no processo de descoberta e aquisição de conhecimento. Ao expor um desenvolvedor a formas alternativas de se tratar exceções, espera-se que esse desenvolvedor torne-se ciente de que é possível tratar uma exceção de maneira mais adequada do que simplesmente imprimir a pilha de execução associada à exceção (do inglês *stack trace*). De maneira otimista, é esperado que esta ciência adquirida seja a etapa inicial no processo de aquisição do conhecimento necessário para a implementação adequada de código de tratamento de exceções.

Sistemas de recomendação, em geral, provêem três funcionalidades: um mecanismo de coleta de dados, um mecanismo de recomendação e uma interface gráfica com o usuário. No sistema de recomendação proposto os esforços foram focados na definição e desenvolvimento dos mecanismos de coleta de dados e de recomendação. Desenvolveu-se apenas uma interface gráfica básica para fins de teste dos mecanismos de coleta de dados e de recomendação. Em trabalhos futuros, será desenvolvida uma interface gráfica mais apropriada para o uso dos usuários finais.

Em um sistema de recomendação, o mecanismo de coleta de dados tem como objetivo coletar e armazenar em uma base de dados informações que sejam relevantes aos usuários. Neste trabalho, um repositório de exemplos foi montado com fragmentos de código previamente implementados e que foram extraídos de projetos de aplicações de software de código aberto. A construção de um repositório de exemplos foi necessária dada a indisponibilidade de repositórios de exemplos indexados com as informações requeridas pelas heurísticas propostas (Seção 3.4.1).

Def.: *Repositório de exemplos* é uma base de dados em que as unidades armazenadas são fragmentos de código. As unidades

armazenadas no repositório de exemplos também são chamadas de candidatos.

A fim de permitir a recuperação dos dados que armazena, o repositório de exemplos é indexado a partir de um conjunto de informações estruturais extraídas de cada fragmento de código armazenado. A este conjunto de informações estruturais é dado o nome de *fatos estruturais*. Os fatos estruturais de um fragmento de código são representados por um conjunto de termos.

Def.: Um *termo* é um par ordenado <C, V> que representa uma informação estrutural (ou fato estrutural) de um determinado fragmento de código. O elemento C é um campo que define um nome para o fato a ser representado. O elemento V é o valor assumido pelo fato estrutural definido pelo campo C no contexto de um fragmento de código.

O mecanismo de recomendação tem como objetivo consultar o repositório de exemplos, selecionar e ranquear candidatos deste repositório e realizar recomendações a um usuário. Neste trabalho, a análise do repositório e a seleção de candidatos é realizada por um conjunto de estratégias heurísticas. Estas heurísticas realizam suas consultas baseadas em fatos estruturais extraídos do código em que o desenvolvedor está trabalhando. Estes fatos estruturais são usados para buscar no repositório de exemplos candidatos que compartilhem similaridades estruturais com o fragmento do desenvolvedor. Os candidatos mais similares são considerados os mais relevantes e são então recomendados ao desenvolvedor.

O desenvolvedor não necessita formular nenhum tipo de consulta (do inglês *query*) para buscar exemplos no repositório. As consultas são criadas automaticamente com base apenas nos fatos estruturais do fragmento de código do desenvolvedor. O processo de criação e execução das consultas é totalmente transparente ao desenvolvedor.

Def.: *Consulta* é a representação de uma necessidade de informação de um usuário em uma linguagem inteligível por um sistema de gerenciamento de bases de dados.

O processo de recomendação ocorre em quatro etapas distintas. Na primeira etapa, o desenvolvedor solicita através de um menu de contexto em seu IDE (do

inglês Integrated Development Environment – Ambiente Integrado de Desenvolvimento) recomendações de tratamento de exceções para o código em que está trabalhando. O sistema de recomendação extrai o método em que o desenvolvedor está trabalhando e o passa para o mecanismo de extração de informações. Na segunda etapa, o mecanismo de extração de informações extrai os fatos estruturais do método em que o desenvolvedor está trabalhando e passa este conjunto de fatos estruturais para o mecanismo de recomendação. Na terceira etapa, o mecanismo de recomendação cria uma única consulta a partir dos fatos estruturais, executa esta consulta no repositório de exemplos e ordena os candidatos retornados pela consulta em ordem de relevância. Na quarta e última etapa, os candidatos retornados pelo mecanismo de recomendação são recomendados ao desenvolvedor. Os detalhes envolvidos em cada etapa serão discutidos nas seções seguintes.

3.3. Mecanismo de coleta de informações

3.3.1. Modelo de dados

Para armazenar as informações coletadas pelo mecanismo de recomendação fez-se necessário definir um modelo de dados para estas informações.

Def.: Um *modelo de dados* descreve um conjunto de dados estruturados a serem armazenados em uma base de dados.

A primeira decisão a respeito do modelo de dados usado pelo mecanismo de coleta de informações foi definir qual a informação que se deseja recomendar ao desenvolvedor (usuário do sistema). Esta informação é conseqüentemente a informação a ser coletada e armazenada. Foi decidido que se deseja recomendar fragmentos de código fonte ao desenvolvedor. Mais especificamente, deseja-se recomendar fragmentos de código que estejam no nível da declaração de um método. Assume-se assim que as informações relevantes ao desenvolvedor estarão contidas no contexto de um fragmento de código que contém a definição de um método, e não espalhadas pelo código de mais de um método. Portanto, a unidade básica de armazenamento definida pelo modelo de dados é um fragmento de código que contem a definição de um método. Vale comentar que poderão ocorrer

casos em que o método contido no fragmento de código (mais especificamente o seu bloco `catch`) faz referência a variáveis de escopo mais abrangente que o método, como, por exemplo, variáveis que são atributos globais em classes. Nestes casos, parte da informação relevante ao desenvolvedor não estará disponível no escopo do exemplo recomendado. Mesmo ciente destes casos de incompletude da informação provida, nesta primeira versão do sistema de recomendação optou-se por manter a granularidade dos exemplos recomendados no nível da declaração de métodos. Considerou-se que a análise das variáveis de escopo mais abrangente dentro do escopo do exemplo é suficiente para o desenvolvedor compreender o seu uso no contexto do tratamento de exceções. Além disso, manter a granularidade das recomendações no nível da declaração de métodos mantém os exemplos de código recomendados concisos. Em versões futuras do sistema de recomendação para código de tratamento de exceções pretende-se explorar o uso de um analisador sintático mais sofisticado. Em particular, um analisador sintático capaz de extrair como exemplo não apenas a declaração de um método, mas também suas dependências com elementos de escopo mais abrangente. Consequentemente, também pretende-se verificar se essas informações extra trazem de fato algum ganho para o desenvolvedor em termos de qualidade e relevância da recomendação realizada.

Definida a granularidade da unidade de informação a ser coletada e armazenada, decidiram-se então quais informações de cada fragmento de código a ser armazenado deveriam ser extraídas a fim de indexá-los. Esta decisão foi guiada pela especificação de cada uma das heurísticas propostas e que serão apresentadas em detalhes na Seção 3.4.1. Para cada fragmento de código armazenado, as seguintes informações adicionais são extraídas:

- Conjunto de tipos das exceções que são tratadas no contexto do método contido no fragmento de código;
- Conjunto dos nomes completamente qualificados (do inglês *fully qualified name*) dos métodos invocados no contexto do método contido no fragmento de código;
- Conjunto de tipos das variáveis usadas no contexto do método contido no fragmento de código.

Definidos a unidade de informação a ser armazenada e o conjunto de informações que devem ser extraídas, chegou-se ao modelo de dados mostrado na Figura 6:

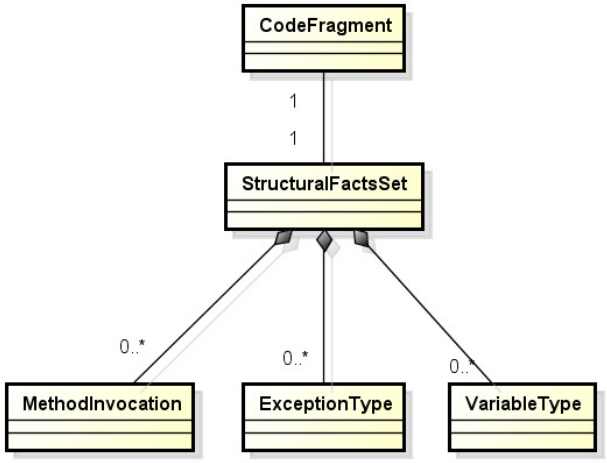

Figura 6 Modelo de dados

Como mostra a Figura 6, cada fragmento de código possui associado a si um conjunto de fatos estruturais. Cada conjunto de fatos estruturais é composto por: um conjunto de métodos invocados no contexto do fragmento de código, um conjunto de tipos de exceções tratadas no contexto do fragmento de código e um conjunto de tipos de variáveis usadas no contexto do fragmento de código.

3.3.2. Extração

O processo de extração dos fragmentos de código a serem armazenados no repositório de exemplos foi automatizado através da implementação de um analisador sintático customizado para esta tarefa. O processo de extração realizado pelo analisador sintático ocorre da seguinte maneira:

```
PARA CADA arquivo Java FAÇA

    Represente o arquivo Java como sua árvore sintática

    PARA CADA nó na árvore sintática FAÇA

           SE nó representa uma declaração de método
           E nó contém um bloco catch FAÇA

                  SE nó contém bloco catch ineficaz ENTÃO

                         Passe para o nó seguinte

                  SENÃO

                         Extraia os fatos estruturais do nó

                         Crie no repositório um registro contendo o
                         código fonte relativo ao nó

                         Use os fatos estruturais para indexar
                         o registro contendo o código fonte
                         relativo ao nó
                  FIM SE
           FIM SE
    FIM PARA
FIM PARA
```

Percorrem-se todos os arquivos Java de um projeto e para cada um deles cria-se uma representação do arquivo em formato de árvore sintática. Uma árvore sintática representa a estrutura do código fonte como uma estrutura de dados árvore. O analisador sintático percorre os nós desta árvore sintática para extrair os fragmentos de código que serão usados como recomendação. Como se deseja recomendar fragmentos de código que implementam tratamento de exceções, o analisador sintático considera apenas os nós correspondentes a uma declaração de método que contém no seu corpo um ou mais blocos `catch`. Além disso, o analisador sintático também verifica se o bloco `catch` contido pela declaração de método corrente é um tratador "ineficaz" (MCCUNE, 1996; CHEN *et al.*, 2009). Exemplos de tratadores considerados "ineficazes" e que são descartados pelo analisador sintático são: blocos `catch` vazios, blocos `catch` que apenas imprimem a pilha de execução da exceção, blocos `catch` que apenas retornam `null`, dentre outros. A fim de possibilitar que as unidades armazenadas sejam consultadas posteriormente, elas são indexadas no momento em que são inseridas na base. Cada fragmento de código armazenado é indexado por um conjunto de termos relativos aos seus fatos estruturais. Os termos usados para indexar os fragmentos de código estão diretamente relacionados com as informações

definidas pelo modelo de dados da Seção 3.3.1. Os campos dos termos usados para indexar os fragmentos de código estão definidos na

Tabela 1:

Tabela 1 Campos usados pelos termos

Nome do campo	Descrição
handles	Este campo indica quais exceções são tratadas em um determinado fragmento de código.
calls	Este campo indica quais métodos são chamados no contexto de um determinado fragmento de código.
uses	Este campo indica quais tipos são usados no contexto de um determinado fragmento de código.

Os campos definidos na

Tabela 1 expõem os fatos estruturais coletados para cada exemplo de código armazenado. A partir destes fatos estruturais, as heurísticas poderão consultar o repositório de exemplos durante a busca por candidatos para a recomendação. Mais detalhes sobre a definição e a concretização das heurísticas serão discutidos nas Seção 3.4.

3.3.3. Fonte de exemplos

As unidades armazenadas no repositório de exemplos são trechos de código contendo declarações de métodos. Tal repositório de exemplos é populado com fragmentos de código que são extraídos de projetos de aplicações de software previamente implementados. Uma dificuldade encontrada para popular o repositório de exemplos foi encontrar as aplicações de onde se extrairiam os exemplos de código. Ainda que existam na Internet diversos portais que hospedam projetos de software de código aberto, como, por exemplo, *SourceForge, Google Project Hosting, GitHub, BitBucket, Apache Software Foundation, Eclipse Foundation Open Source Community,* dentre outros, não são quaisquer projetos que servem como fonte de exemplos. Partiu-se da premissa de que quanto melhor as informações disponíveis, melhores serão as recomendações realizadas. Fez-se necessário então buscar por aplicações de software com código

de tratamento de exceções minimamente bem estruturado para que fossem usadas como fonte de exemplos.

Inicialmente, optou-se então por usar apenas aplicações hospedadas pela *Eclipse Foundation Open Source Community.* Esta opção foi tomada devido, principalmente, à reconhecida qualidade dos projetos hospedados neste portal. Além disso, outros trabalhos propondo sistemas de recomendação também se baseiam em aplicações baseadas no Eclipse (MANDELIN *et al.*, 2005; HOLMES, WALKER e MURPHY, 2006; BRUCH, MEZINI e MONPERRUS, 2010). Desta forma, futuramente será possível realizar comparações mais apropriadas entre o sistema de recomendação proposto nesta dissertação e esses trabalhos. Ainda que as aplicações hospedadas no portal do Eclipse sejam todas baseadas em um mesmo *framework*, elas apresentam propósitos bastante distintos. Esta heterogeneidade das aplicações possibilita a realização de recomendações relevantes em diversos cenários de tratamento de exceções. Outro fator que influenciou esta decisão foi o fato de a *Eclipse Foundation Open Source Community* prover documentação razoável e completa para estas aplicações, além de fóruns, listas e canais de bate-papo em que podem ser encontrados tanto os desenvolvedores do *framework* Eclipse, quanto os desenvolvedores das aplicações. Desta forma, o levantamento de informações extras que seriam úteis nas avaliações do sistema de recomendação proposto seria mais fácil.

Para garantir a qualidade mínima do código das aplicações selecionadas, alguns critérios foram estabelecidos:

- O código fonte da aplicação deveria estar disponível em um repositório sob o domínio da *Eclipse Foundation Open Source Community;*
- A aplicação deveria ser categorizada como madura pela *Eclipse Foundation Open Source Community*[7].

Os dois critérios estabelecidos garantem que a aplicação tenha uma qualidade mínima, visto que a *Eclipse Foundation Open Source Community* apenas admite como projeto "Maduro" aqueles que passam pelo crivo de um comitê de qualidade. Além destes critérios, uma inspeção do código de tratamento

[7] Os projetos hospedados pela Eclipse Foundation Open Source Community são classificados em "Maduro" ou "Em Incubação".

de exceções de algumas aplicações foi realizada com o auxílio do analisador sintático responsável por extrair os exemplos das aplicações.

O código fonte das aplicações foi baixado manualmente de seus sistemas de controle de versões e analisados com uma versão modificada do analisador sintático que será descrito em detalhes na Seção 4.1.1. Ao invés de inserir os exemplos de código no repositório de exemplos, a versão modificada do analisador sintático salva o código fonte em arquivo texto para posterior análise manual. Dado o grande número de exemplos disponíveis (cerca de 20.000 exemplos), a análise foi feita por amostragem. Para cada aplicação, um conjunto de vinte exemplos (correspondente a 1% do total) foi escolhido aleatoriamente e cada exemplo era avaliado quanto à qualidade de seu tratador. Exemplos implementando tratadores ineficazes definidos em (MCCUNE 2006; CHEN *et al.*, 2007) eram considerados automaticamente de baixa qualidade. Outros tratadores excessivamente triviais, como apenas registrar em arquivo de *log* a exceção, apenas retornar um determinado valor, ou apenas testar assertivas, também eram considerados automaticamente de baixa qualidade. Caso uma aplicação apresentasse uma amostra com pelo menos dez tratadores de baixa qualidade, esta aplicação era descartada. Ao fim da inspeção, as aplicações abaixo foram então usadas como fonte de exemplos:

- Parallel Tools Platform
- EJB Tools
- JEE Tools
- Object Teams
- Web Services Tools
- Buckminster Component Assembly
- C/C++ Development Tooling
- JDT Core
- JDT Debug
- JDT Ui
- Eclipse Platform Team

Ao todo, foram extraídas 7919 declarações de métodos. Vale salientar o fato de que análises da qualidade de tratadores de exceções já haviam sido realizadas em trabalhos preliminares desta dissertação (Seção 2.3), além de estudos

realizados em colaboração com pesquisadores da Universidade Federal do Rio Grande do Norte, e que as aplicações usadas como fonte de exemplos só foram escolhidas por apresentarem tratadores consideravelmente mais bem estruturados do que os apresentados pelas aplicações analisadas nesses trabalhos.

3.4. Mecanismo de recomendação

As heurísticas constituem o cerne do mecanismo de recomendação. Elas são responsáveis por buscar, selecionar e ranquear os candidatos do repositório de exemplos antes que estes sejam recomendados ao desenvolvedor. As heurísticas definidas neste trabalho baseiam suas buscas em fatos estruturais do código fonte. A partir dos fatos de um determinado fragmento de código, as heurísticas buscam candidatos que compartilhem com este fragmento similaridades estruturais. A premissa das heurísticas é que ao recomendar fragmentos de código similares há maior chance de o código de tratamento de exceções recomendado ser mais relevante no contexto do desenvolvedor. Nesta dissertação, foram definidas três diferentes heurísticas de busca, cada uma baseada em fatos estruturais distintos. Os fatos estruturais explorados pelas heurísticas estão relacionados a informações como: o tipo da exceção que é tratada, os métodos invocados e os tipos das variáveis usadas no escopo da definição de um método. Ainda que outras informações básicas pudessem ser exploradas, nesta primeira versão do sistema de recomendação para código de tratamento de exceções optou-se por usar um conjunto conciso de fatos estruturais. Optou-se por manter o conjunto de fatos estruturais conciso a fim de explorar o uso de estratégias heurísticas que fossem simples de implementar, isto é, estratégias heurísticas que fossem baseadas em informações estáticas que pudessem ser extraídas do código fonte sem demandar muito esforço computacional. Mesmo conciso, o conjunto de fatos estruturais usado pelas heurísticas acabou mostrando-se suficiente para identificar similaridades estruturais, como será discutido na Seção 5.1.1. Em versões futuras do sistema de recomendação, pretende-se explorar outros fatos estruturais a fim de verificar se há ganhos na relevância dos exemplos recomendados. Outros exemplos de informações a serem futuramente exploradas são os identificadores das variáveis usadas, o tipo da classe que a classe que contém a declaração do

método está estendendo, ou se a declaração do método sobrescreve algum método da classe mãe. Nas seções a seguir cada uma das heurísticas é apresentada e definida.

3.4.1. Heurísticas de busca

3.4.1.1. Heurística I - Tipo da Exceção Tratada

Motivação. Linguagens de programação que provêem mecanismos de tratamento de exceções embutidos na própria linguagem tipicamente definem blocos tratadores de exceção através da associação de um tipo de exceção a um conjunto de instruções. Estas instruções têm por objetivo recuperar o estado do sistema da ocorrência de uma exceção desse tipo. Também é comum nestas linguagens estruturar os tipos de exceções hierarquicamente em árvores de tipos. Idealmente, os tipos de exceções que representam situações excepcionais similares devem estar agrupados em uma mesma sub-árvore. Neste contexto, a *Heurística Tipo da Exceção Tratada* assume que tratadores com tipos de exceção proximamente relacionadas na hierarquia de tipos têm maior chance de serem mais similares.

Fatos necessários. A heurística do tipo da exceção tratada necessita de informações relativas ao (i) tipo da exceção que se deseja tratar e também de informações a respeito da (ii) hierarquia de tipos de exceções. Assume-se que os tipos de exceções que se desejam tratar são todos os tipos que podem ser levantados no contexto do método que o desenvolvedor está trabalhando.

Definição. Dado um método m_1 em que se deseja tratar uma exceção do tipo T_1, a heurística "Hierarquia do tipo da exceção tratada" busca por candidatos m_2 que contenham estruturas `try-catch` ou `try-catch-finally` baseado no tipo da exceção declarada em seu(s) bloco(s) `catch`. A *Heurística Hierarquia do Tipo da Exceção Tratada* buscará candidatos m_2 com blocos `try-catch` ou `try-catch-finally` que declarem no bloco `catch` uma exceção `T2` que seja:

I. T_2 é do mesmo tipo que T_1;

II. T_2 é supertipo de T_1, desde que T_2 não seja um dos tipos mais genéricos da linguagem.

Optou-se por também considerar os candidatos que implementam tratadores cujas exceções são supertipo da exceção que se deseja tratar devido aos casos em que um tratador trata uma exceção por subsunção. Assume-se assim que mesmo em um tratador mais genérico do que o desejado podem existir informações relevantes para o tratamento de uma determinada exceção. Ademais, consideram-se os candidatos que satisfazem a condição I mais relevantes do que os candidatos que satisfazem apenas a condição II. A Figura **7** a seguir exemplifica o funcionamento da *Heurística Tipo da Exceção Tratada*.

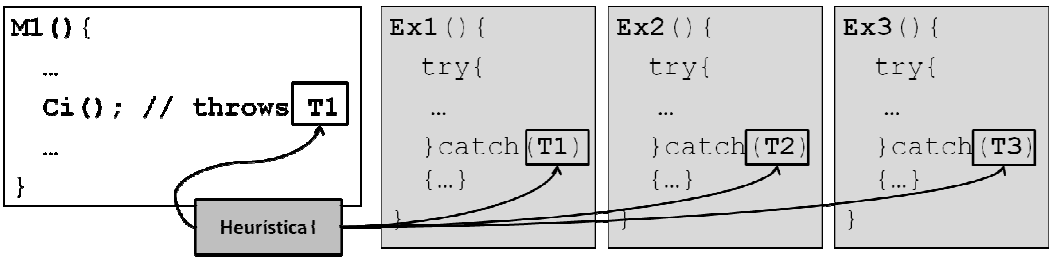

Figura 7 Exemplo da Heurística Tipo da Exceção Tratada

Considere o método `M1` na Figura **7** como sendo o método que o desenvolvedor está implementando. O método `M1` invoca o método `Ci` que, por sua vez, lança uma exceção do tipo `T1`. Considere também que o desenvolvedor deseja tratar a exceção do tipo `T1` lançada por `Ci` no corpo do método `M1`. A *Heurística Tipo da Exceção Tratada* irá examinar exemplos de código implementando tratamento de exceções: métodos `Ex1`, `Ex2` e `Ex3` da Figura **7**. Considere ainda que o tipo de exceção `T2` é um tipo ascendente de `T1` na hierarquia de tipos de exceções e que `T3` não é ascendente de `T1`. No cenário exemplificado na Figura **7**, os exemplos `Ex1` e `Ex2` são selecionados, enquanto o exemplo `Ex3` é desconsiderado. O exemplo `Ex1` é selecionado porque trata uma exceção do tipo `T1`, ou seja, `Ex1` trata uma exceção com o mesmo tipo que se deseja tratar em `M1`. O exemplo `Ex2` é selecionado porque trata uma exceção que é ascendente de `T1`, ou seja, `Ex2` trata uma exceção que é super-tipo da exceção que se deseja tratar em `M1`. O exemplo `Ex3` não é selecionado porque trata uma exceção que não é ascendente de `T1`, ou seja, `Ex3` trata uma exceção que não se relaciona na hierarquia de tipos de exceções com o tipo da exceção que se deseja

tratar em M1. Dentre os exemplos selecionados, o exemplo Ex1 trata uma exceção que tem o mesmo tipo da exceção que o desenvolvedore deseja tratar, enquanto o exemplo Ex2 trata uma exceção que é apenas um super tipo da exceção que o desenvolvedor deseja tratar em M1. Por este motivo, o exemplo Ex1 é considerado pela *Heurística Tipo da Exceção Tratada* mais relevante do que o exemplo Ex2. Desta forma, o exemplo Ex1 é recomendado em uma melhor posição na lista de recomendações do que o exemplo Ex2.

3.4.1.2. Heurística II - Seqüência de Chamadas

Motivação. Definir o tratamento adequado de uma exceção não depende apenas do tipo da exceção que está sendo tratada, mas também do contexto do método onde ocorre o tratamento. A fim de tentar representar o contexto de um método, a *Heurística Seqüência de Chamadas* assume que métodos que realizam seqüências de chamadas similares possuem contexto similar e, portanto, possuem maior chance de implementarem tratadores de exceções mais similares.

Fatos necessários. A heurística da seqüência de chamadas necessita apenas do conjunto de chamadas de métodos que ocorrem no contexto do método sob implementação.

Definição. Dado um método m_1 que realiza uma seqüência de chamadas a métodos c_0 ... c_n, que em alguma chamada c_i (i=0...n) lança a exceção do tipo T_1 e deseja-se tratar T_1. A *Heurística Sequência de Chamadas* busca por candidatos baseada na seqüência de chamadas c_0 ... c_n feitas por m_1. A heurística "Seqüência de chamadas" buscará por candidatos m_2 que realizem obrigatoriamente alguma chamada a c_i. Além disso, quanto maior o número de chamadas comuns entre m_1 e m_2, mais relevante m_2 será considerado. A Figura **8** exemplifica o funcionamento da *Heurística Sequência de Chamadas*.

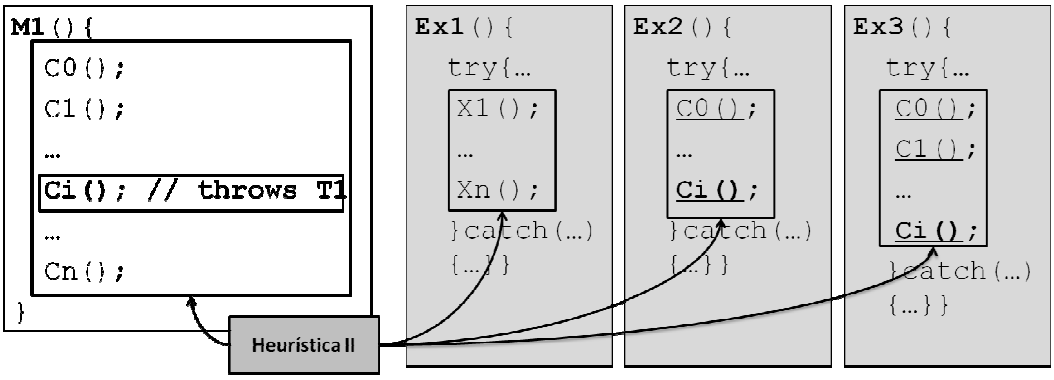

Figura 8 Exemplo da Heurística Sequência de Chamadas

Considere o método `M1` mostrado na Figura **8** como sendo o método que o desenvolvedor está implementando. O método `M1` invoca em seu corpo o método `Ci` que, por sua vez, lança uma exceção do tipo `T1`. Considere neste cenário que o desenvolvedor deseja tratar a exceção lançada por `Ci` no corpo do método `M1`. Neste cenário, os exemplos disponíveis são os exemplos `Ex1`, `Ex2` e `Ex3` mostrados na Figura **8**. A *Heurística Sequência de Chamadas* irá selecionar os exemplos a serem recomendados baseada nos métodos invocados por `M1`. Os exemplos `Ex2` e `Ex3` são selecionados, pois ambos invocam o método `Ci`. O exemplo `Ex1` é descartado, pois não invoca o método `Ci`. Denre os exemplos selecionados, o exemplo `Ex3` invoca três métodos (`C0`, `C1` e `Ci`) que também são invocados pelo método `M1`, enquanto o exemplo `Ex2` invoca dois métodos (`C0` e `Ci`) que também são invocados pelo método `M1`. Desta forma, a *Heurística Sequência de Chamadas* considera o `Ex3` mais relevante do que o `Ex2` e, por isso, recomenda `Ex3` em uma posição melhor do que `Ex2` na lista de recomendações.

3.4.1.3. Heurística III - Tipos das Variáveis

Motivação. A alguns tipos de objetos geralmente estão associados tipos específicos de exceções. Por exemplo, Desta forma, a *Heurística Tipos das Variáveis* assume que métodos que usam (declaram, referenciam ou recebem como parâmetro) variáveis do mesmo tipo têm maior chance de serem similares e, portanto, de possuírem tratadores mais relevantes.

Fatos necessários. A heurística dos tipos das variáveis necessita apenas do conjunto dos nomes dos tipos das variáveis usadas no contexto do método sob implementação.

Definição. Dado um método m_1 em que se deseja tratar a exceção do tipo T_1. Considere V_1 o conjunto de variáveis que m_1 usa em seu corpo, e TV_1 o conjunto de tipos das variáveis pertencentes a V_1. A *Heurística Tipos das Variáveis* buscará por métodos m_2 que usem ao menos uma variável cujo tipo pertença a TV_1. Quanto maior o número de variáveis usadas em m_2 e que pertençam a TV_1, mais relevante m_2 será considerado. O cenário mostrado na Figura **9** exemplifica o funcionamento da *Heurística Tipos das Variáveis*.

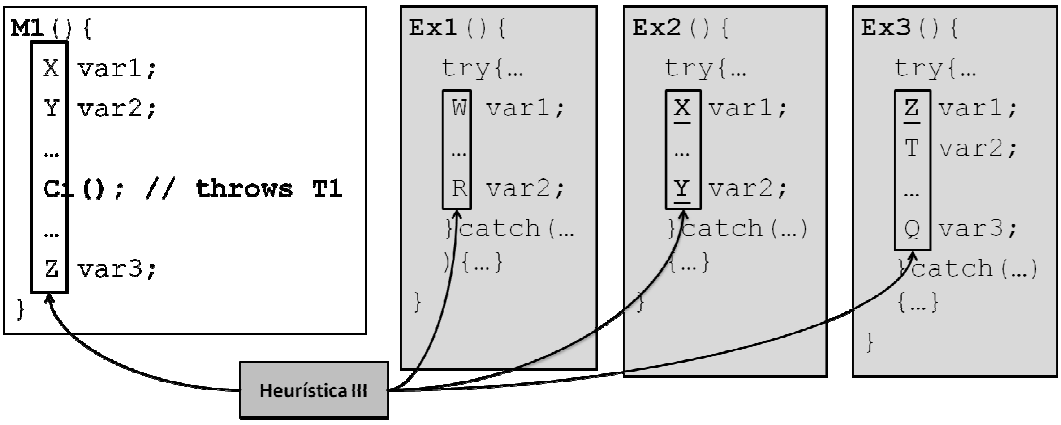

Figura 9 Exemplo da Heurística Tipos das Variáveis

Considere o método `M1` da Figura **9** como o método que o desenvolvedor está implementando. O método `M1` invoca o método `Ci` que, por sua vez, lança a exceção `T1`. Considere que o desenvolvedor deseja tratar a exceção `T1` no corpo de `M1`. No cenário exemplificado na Figura **9**, os exemplos disponíveis para recomendação são os métodos `Ex1`, `Ex2` e `Ex3`. A *Heurística Tipos das Variáveis* irá selecionar os exemplos a serem recomendados baseados no conjunto de tipos das variáveis usadas no corpo do método `M1`. Neste cenário, o conjunto de tipos das variáveis de `M1` é `{X, Y, Z}`. Os exemplos `Ex2` e `Ex3` são selecionados, pois ambos usam ao menos uma variável cujo tipo também é usado por `M1`. O exemplo `Ex1` não é selecionado, pois não usa qualquer variável cujo tipo também seja usado por `M1`. Dentre os exemplos selecionados, o exemplo `Ex2` possui mais tipos de variáveis em comum (tipos `X` e `Y`) com o método `M1` do que o exemplo `Ex3` (apenas o tipo `Z`). Desta forma, a *Heurística Tipos das Variáveis*

considera o exemplo `Ex2` mais relevate do que o exemplo `Ex3` e, por isso, `Ex2` é recomendado em uma melhor posição na lista de recomendações do que `Ex3`.

3.4.2. Concretização das heurísticas: consultas e pontuação

Como descrito na Seção 3.4.1, as heurísticas realizam suas buscas no repositório de exemplos a partir de fatos estruturais. Estes fatos, por sua vez, são extraídos do fragmento de código em que o desenvolvedor está trabalhando. Além de buscar e selecionar candidatos no repositório, as heurísticas também devem ser capazes de classificar os candidatos selecionados em ordem de relevância. Desta forma, a concretização das heurísticas requer:

- Um mecanismo de consulta ao repositório de exemplos com base nos fatos estruturais extraídos de um fragmento de código;
- Um mecanismo de pontuação que confira aos candidatos selecionados uma pontuação relativa ao seu grau de relevância de acordo com os fatos estruturais extraídos do fragmento de código.

A Seção 3.4.2.1 descreve o mecanismo de consultas desenvolvido para o sistema de recomendações, detalhando como ele cria consultas a partir dos fatos estruturais extraídos e como as executa no repositório a fim de selecionar exemplos a serem recomendados. A Seção 3.4.2.2 descreve o mecanismo de pontuação, detalhando a sua função que atribui a um par `<candidato, consulta>` um valor real proporcional à relevância do candidato em relação à consulta.

3.4.2.1. Mecanismo de consultas

As heurísticas têm como função primordial selecionar candidatos no repositório de exemplos com base nos fatos estruturais extraídos do código do desenvolvedor. Para concretizar tais heurísticas, é necessário então criar consultas baseadas nos fatos estruturais. Desta forma é possível consultar o repositório de exemplos a fim de encontrar candidatos que ajudem o desenvolvedor a descobrir como tratar uma exceção no contexto de seu fragmento de código. O mecanismo

de consulta é o responsável por construir essas consultas que orientam a busca no repositório de exemplos.

As consultas construídas pelo mecanismo de consulta são baseadas em um modelo booleano de recuperação de informações (MANNING, RAGHAVAN e SCHÜTZE, 2008), isto é, elas são representadas como expressões booleanas de termos. O modelo booleano para recuperação de informações apenas identifica se um determinado candidato satisfaz ou não uma consulta. Nenhum tipo de pontuação de relevância de candidatos é realizado e os candidatos selecionados não são ordenados. Uma consulta simplesmente identifica um subconjunto de candidatos que a satisfazem. A pontuação dos candidatos selecionados pelas consultas é discutida na Seção 3.4.2.2.

Apesar de terem sido definidas separadamente na Seção 3.4.1, todas as heurísticas são realizadas como uma única consulta. Desta forma é possível recuperar todos os candidatos selecionados consultando o repositório apenas uma vez. A consulta única é construída da seguinte maneira: seja `FATOS` o conjunto de termos que representam os fatos estruturais extraído de um fragmento de código. A consulta `Q` baseada em `FATOS` é definida pela expressão booleana disjuntiva (expressão booleana em que todos os termos são conectados pelo operador `OU`) de cada um dos termos em `FATOS`:

$$Q = \bigvee_{t:Termo \in FATOS} t$$

Considere como exemplo o seguinte conjunto de termos extraído de um fragmento de código:

```
FATOS =
{
    handles:FileNotFoundException,
    calls:openFile,
    calls:readFile,
    calls:closeFile,
    uses:File,
    uses:FileWriter,
    usesBufferedWriter
}
```

É importante lembrar que, como definido na especificação da heurística do tipo da exceção tratada (Seção 3.4.1.1), há a necessidade de também se buscar por exemplos de código que tratam exceções que são supertipo da exceção que se deseja tratar. Ou seja, neste exemplo é necessário buscar por exemplos de código

que tratem exceções que sejam supertipos de `FileNotFoundException`. Desta forma, para a construção das consultas, consideram-se todos os tipos ascendentes, que não seja um dos tipos mais genéricos da linguagem, do tipo de exceção definido no conjunto de fatos estruturais. Portanto, a consulta `Q` para o conjunto de fatos estruturais `FATOS` construída pelo mecanismo de consulta terá a forma:

```
Q =
    handles:FileNotFoundException OR
    handles:IOException OR
    calls:openFile OR
    calls:readFile OR
    calls:closeFile OR
    uses:File OR
    uses:FileWriter OR
    uses:BufferedWriter
```

No exemplo anterior, como o tipo `IOException` é pai do tipo `FileNotFoundException` acrescentou-se a consulta `Q` o termo `handles:IOException`. Visto que o pai direto do tipo `IOException` é o tipo `Exception`, um dos tipos mais genéricos da linguagem, o termo `handles:Exception` não aparece na consulta resultante.

As consultas construídas pelo mecanismo de consultas são representadas como expressões booleanas disjuntivas. A representação das consultas como expressões booleanas conjuntivas, em que todos os termos são conectados pelo operador `E`, também é possível. Entretanto, esta seria uma representação muito restritiva, selecionando poucos candidatos, visto que todos os termos teriam que ser satisfeitos para que um candidato fosse selecionado por uma consulta. A forma disjuntiva é a forma menos restritiva das possíveis, ou seja, é a forma que seleciona o maior número de candidatos. Dado que todas as heurísticas são representadas como uma consulta única, a forma disjuntiva permite que um candidato seja selecionado mesmo que não satisfaça os critérios de todas as heurísticas, mesmo que ao custo de muitos candidatos serem selecionados. O grande número de candidatos selecionados por esta representação das consultas é tratado através da ordenação dos candidatos segundo uma função de pontuação, a qual atribui a cada candidato uma nota proporcional a sua relevância para com a consulta.

3.4.2.2. Mecanismo de pontuação

O mecanismo de consultas apresentado na Seção 3.4.2.1 apenas seleciona dentre os candidatos disponíveis no repositório de exemplos quais satisfazem a uma determinada consulta. O número de candidatos que satisfazem uma determinada consulta pode facilmente ultrapassar o número de centenas. Recomendar todos estes candidatos selecionados pelo mecanismo de consulta de maneira não ordenada diretamente para o desenvolvedor seria ineficaz. Muito provavelmente o desenvolvedor se perderia em meio a tanta informação e não conseguiria satisfazer a sua necessidade de informação. Por este motivo, é vital que antes de realizar uma recomendação ao desenvolvedor os candidatos selecionados sejam ordenados de acordo com um grau de relevância a uma determinada consulta. Depois de ordenados, apenas os candidatos com maior grau de relevância são recomendados ao desenvolvedor.

A fim de graduar os candidatos quanto à sua relevância frente a uma consulta, definiu-se uma *função de pontuação*. A função de pontuação associa a um candidato do repositório e a uma determinada consulta um valor real que representa o grau de relevância daquele candidato em relação àquela consulta. É com base neste valor real que os candidatos são ordenados: os candidatos com maior nota são considerados mais relevantes, e por isso ocupam as primeiras posições na lista de recomendação, enquanto os candidatos com menores notas são considerados menos relevantes, ocupando as últimas posições.

A função de pontuação baseia-se em dois fatores. O primeiro fator da função de pontuação baseia-se na quantidade de termos da expressão booleana que representa uma consulta que são satisfeitos por um candidato. É dito que um candidato satisfaz um termo de uma expressão booleana quando este candidato possui um fato estrutural equivalente a um dos termos da consulta. Em termos práticos, um candidato satisfaz um termo de uma expressão booleana quando é indexado no repositório por um termo `campo:valor` igual a um dos termos da expressão. O primeiro fator da função de pontuação define então que candidatos que satisfazem mais termos de uma expressão booleana recebem maior pontuação do que candidatos que satisfazem menos termos. O objetivo do primeiro fator da função de pontuação é identificar candidatos que têm maior relevância em relação

a uma determinada consulta. Consequentemente, também são identificados os candidatos mais similares com o fragmento de código em que o desenvolvedor está trabalhando, visto que a consulta é construída baseada em seus fatos estruturais. Desta forma, dizer que um candidato `C1` satisfaz mais termos de uma consulta `Q` do que um candidato `C2` significa que o candidato `C1` possui mais fatos estruturais similares ao fragmento de código em que o desenvolvedor está trabalhando do que o candidato `C2`.

Para calcular o primeiro fator da função de pontuação, a função `S` foi definida. A função `S` calcula para um candidato `C` e uma consulta `Q` a relação entre:

- Numerador – a quantidade de termos da consulta `q` satisfeitas pelo candidato `c`. Ou dito de outra forma: a quantidade de termos da consulta `q` que pertencem aos fatos estruturais do candidato `c`.
- Denominador – quantidade total de termos da consulta `q`.

$$S(c:\text{Candidato}, q:\text{Consulta}) = \frac{|t:\text{Termo} \mid t \in q \wedge t \in c.\text{Fatos}|}{|t:\text{Termo} \mid t \in q|}$$

Considere os fatos estruturais de dois candidatos `C1` e `C2` dados, respectivamente, por:

```
FACTS_1 = {handles:IOException, calls:openFile, calls:readData,
calls:closeFile, uses:File, uses:BufferedReader }

FACTS_2 = { handles:SQLException, uses:Resource, uses:Connection }
```

E considere a seguinte consulta `Q`:

```
Q = handles:IOException OR calls:getPath OR uses:Resource
```

A função `S` calculará os seguintes valores para cada par:

```
S( C1, Q ) = | {handles:IOException} | /
           | {handles:IOException, calls:getPath, uses:Resource } |
S( C1, Q ) = 1 / 3 = 0.33

S( C2, Q ) = | { uses:Resource, uses:Connection } | /
             | {handles:IOException, calls:getPath, uses:Resource} |
S( C1, Q ) = 2 / 3 = 0.67
```

No exemplo acima o candidato `C1` satisfaz apenas um termo da consulta `Q` (`handles:IOException`), enquanto o candidato `C2` satisfaz dois termos (`uses:Resource` e `uses:Connection`). Desta forma, segundo a função `S`,

o candidato C2 foi considerado mais relevante em relação à consulta Q por possuir mais termos satisfeitos.

O segundo fator da função de pontuação atribui um peso ajustável a cada um dos campos que compõem os termos da expressão booleana: (uses, handles, calls). Desta forma, podem ser definidos pesos com valores diferentes para cada um dos campos usados nos termos a fim de atribuir relevâncias distintas para cada um dos campos. Os pesos podem ser ajustados a cada sessão de uso do sistema de recomendação. Isto quer dizer que é possível definir que um campo é mais relevante do que outro de acordo com o cenário de uso. Pode-se definir que em determinado cenário a satisfação de um termo com campo calls é mais relevante do que a satisfação de um termo com o campo uses, por exemplo.

O segundo fator da função de pontuação é definido pela seguinte função P:

$$P(c:\text{Candidato}, q:\text{Consulta}) = \sum_{t:\text{Termo} \quad t \in q \wedge t \in c.\text{Fatos}} \text{PESO}(t.\text{campo})$$

A função P calcula o somatório dos pesos dos campos dos termos da consulta Q que são satisfeitos pelo candidato C. O peso de um campo é definido pela função PESO:

$$PESO(f:\text{Campo}) = K$$

Em que K é definido pelo desenvolvedor do sistema de recomendações para cada um dos campos usados nos termos das consultas.

Considere então os mesmos candidatos C1 e C2 e a mesma consulta Q do exemplo anterior. E seja a função PESO definida da seguinte forma para cada campo:

```
PESO ( handles ) = 5.0
PESO ( uses ) = 1.0
PESO ( calls ) = 1.0
```

A função P calculará os seguintes valores para cada par:

```
P(C1, Q) = SOMATÓRIO PESO( T.campo ) |
                        T ∈ { handles:IOException }
P(C1, Q) = PESO( handles ) = 5.0

P(C2, Q) = SOMATÓRIO PESO( T.campo ) |
                        T ∈ { uses:Resource, uses:Connection }
P(C2, Q) = PESO( uses ) + PESO( uses ) = 1.0 + 1.0 = 2.0
```

Neste cenário, em que se definiu um peso maior para o campos `handles`, o candidato `C1` obteve uma maior nota do que o candidato `C2`, ainda que `C2` satisfaça um maior número de termos em relação a `Q`. Perceba que o somatório é realizado para cada termo satisfeito, e não para cada campo único. Por este motivo, soma-se `PESO( uses )` duas vezes no cálculo de `P(C2, Q)`.

Finalmente, a função de pontuação `PT` que atribui um valor real proporcional à relevância de um candidato `C` em relação a uma consulta `Q` é definida da seguinte maneira:

$$NOTA(c{:}Candidato, q{:}Consulta) = S(c,\ q) * P(c,\ q)$$

A função de pontuação `NOTA` é definida como o produto da função S vezes a função P. Desta forma, combinam-se dois fatores de naturezas distintas: uma natureza inerente ao conteúdo dos exemplos do repositório e ao conteúdo do código sob implementação e outra que pode ser manipulada pelo usuário. O primeiro fator é responsável por encontrar os candidatos que compartilham similaridades estruturais com o código sob implementação. O segundo fator permite ao desenvolvedor-usuário do sistema de recomendação refinar as recomendações de acordo com suas necessidades. Este refinamento pode, inclusive, ser realizado a cada sessão de uso do sistema.

Assumindo os mesmos valores para `C1`, `C2`, `Q`, `PESO( handles )`, `PESO( calls )` e `PESO( uses )` usados no exemplo anterior, os valores calculados por `NOTA` seriam os seguintes:

```
NOTA(C1, Q) = S( C1, Q ) * P( C1, Q )
NOTA(C1, Q) = 0.33 * 5.0 = 1.65

NOTA(C2, Q) = S( C2, Q ) * P( C2, Q )
NOTA(C2, Q) = 0.66 * 2.0 = 1.32
```

Neste exemplo, o candidato `C1` recebeu uma nota mais alta do que o candidato `C2`, o que significa que `C1` foi considerado mais relevante do que `C2`

pelo mecanismo de pontuação. Mesmo que para a consulta Q o candidato C2 tenha satisfeito mais fatos estruturais do que o candidato C1, o uso dos pesos atribuídos aos campos fez com que o candidato C1 recebesse uma maior nota de relevância. Em outro cenário, caso o desenvolvedor-usuário do sistema de recomendação alterasse o valor dos pesos, este resultado poderia ser diferente. O ajuste de pesos provê aos desenvolvedores-usuários do sistema de recomendação flexibilidade para fazer um ajuste fino das recomendações realizadas de acordo com suas necessidades. Mais detalhes a respeito dos ajustes dos pesos são discutidos na Seção 5.1.2.