5
Avaliação

O Capítulo 4 apresentou os detalhes de implementação do sistema de recomendações para código de tratamento de exceções proposto. Neste capítulo, serão apresentados as avaliações realizadas sobre a implementação do sistema. O restante do capítulo está estruturado da seguinte maneira. Na Seção 5.1 discutem-se as avaliações preliminares que foram realizadas já durante o processo de desenvolvimento do sistema de recomendações. Já na Seção 5.2 discutem-se avaliações mais detalhadas, de caráter predominantemente qualitativo, realizadas depois de finalizada a implementação do sistema. Por fim, na Seção 5.3 compara-se o sistema de recomendação proposto com trabalhos relacionados.

5.1.
Avaliação preliminar

As avaliações preliminares realizadas durante o processo de desenvolvimento do sistema de recomendações tiveram caráter exploratório-quantitativo. O objetivo destas avaliações preliminares era obter, ainda em fase de desenvolvimento, retroalimentação a respeito das seguintes questões que guiaram a implementação do sistema de recomendações:

Q1. Os fatos estruturais capturados no modelo de dados são suficientes para identificar similaridades estruturais entre fragmentos de código implementando tratamento de exceções?

Q2. A configuração dos pesos ajustáveis associados às heurísticas permite identificar com maior precisão fragmentos de código implementando tratamento de exceções estruturalmente similares?

A fim de colher dados que auxiliassem a resposta destas questões, utilizaram-se experimentos analíticos. A principal característica desejada para os experimentos realizados foi o rápido provimento de dados. Para alcançar este objetivo e facilitar a replicação dos experimentos, implementaram-se programas

de apoio. Tais programas geram dados a respeito da execução do sistema sob análise, que neste caso é o sistema de recomendações, e têm baixo custo de implementação e execução.

O experimento analítico realizado fundamentou-se em um experimento bastante comum em sistemas de recomendações, o experimento *leave-one-out*. Em um experimento *leave-one-out* elimina-se um elemento de um conjunto de dados e verifica-se se o sistema de recomendações sob análise é capaz de recomendar o elemento removido de volta. O experimento analítico usado foi adaptado de (HOLMES, WALKER e MURPHY, 2006) e é configurado da seguinte maneira:

```
INÍCIO Experimento

    PARA CADA candidato no repositório de exemplos FAÇA

           Extraia os fatos estruturais do candidato

           Construa uma consulta com estes fatos estruturais

           Execute a consulta no repositório

           Registre em qual posição o candidato original
           foi recomendado

    FIM PARA

    Retorne o registro de posições em que os candidatos
    foram encontrados

FIM Experimento Base
```

O experimento extrai de um fragmento de código armazenado no repositório de exemplos seus fatos estruturais e usa estes fatos para realizar uma consulta ao repositório. Em seguida, verificam-se os candidatos recomendados, registrando em qual posição o fragmento de código original foi retornado. Optou-se por verificar uma correspondência exata entre o fragmento de código original e os candidatos recomendados, ao invés de verificar a relevância das informações contidas em cada candidato recomendado. Esta opção foi tomada a fim de tornar a análise humanamente tratável e facilmente replicável, visto que durante o processo de desenvolvimento seria necessário executar este experimento diversas vezes. É sabido que a correspondência exata entre o fragmento de código original e um dos candidatos recomendados não simula o cenário real em que o sistema de recomendações será usado.

Entretanto, ao analisar a correspondência exata entre o fragmento original e os fragmentos recomendados tem-se uma espécie de análise do pior caso. Para avaliar a capacidade de identificação de similaridades estruturais, foi verificado se as heurísticas eram capazes de recomendar o fragmento original entre as 10 primeiras recomendações. Caso as heurísticas não fossem capazes em um número razoável de cenários, concluir-se-ia que o modelo de dados e as heurísticas falharam em identificar fragmentos de código estruturalmente similares. Seria necessário então rever a definição e o projeto do modelo de dados e das heurísticas. Esta avaliação, ainda que grosseira quanto à precisão e à relevância das recomendações, teve como grande vantagem a rapidez com que provia retroalimentação. Tal rapidez foi fundamental para que essa avaliação fosse usada durante o processo de desenvolvimento do sistema de recomendação.

A partir da implementação dos programas auxiliares à execução do experimento, configuraram-se dois cenários de execução a fim de auxiliar a resposta de cada uma das questões anteriores. O primeiro cenário está relacionado com a questão Q1, referente à adequação do modelo de dados utilizado. Sua configuração e análise são discutidas na Seção 6.1.2. O segundo cenário está relacionado com a questão Q2, referente ao uso dos pesos ajustáveis das heurísticas utilizadas. Tal cenário será discutido na Seção 5.1.3.

5.1.1. Avaliação da suficiência dos fatos estruturais

A primeira preocupação durante o desenvolvimento do sistema de recomendação implementado foi verificar se o modelo de dados proposto era suficiente para identificar fragmentos de código implementando código de tratamento de exceções que fossem estruturalmente similares. Esta verificação é fundamental já que as heurísticas buscam por candidatos similares baseadas nos fatos estruturais. A fim de se obter uma primeira impressão a respeito da suficiência dos fatos estruturais, executou-se o experimento base definido na Seção 5.1 sem alteração alguma. Com base no registro das posições em que cada candidato do repositório foi encontrado, construiu-se um histograma para avaliar a distribuição das posições. O histograma da Figura 14 mostra os resultados obtidos.

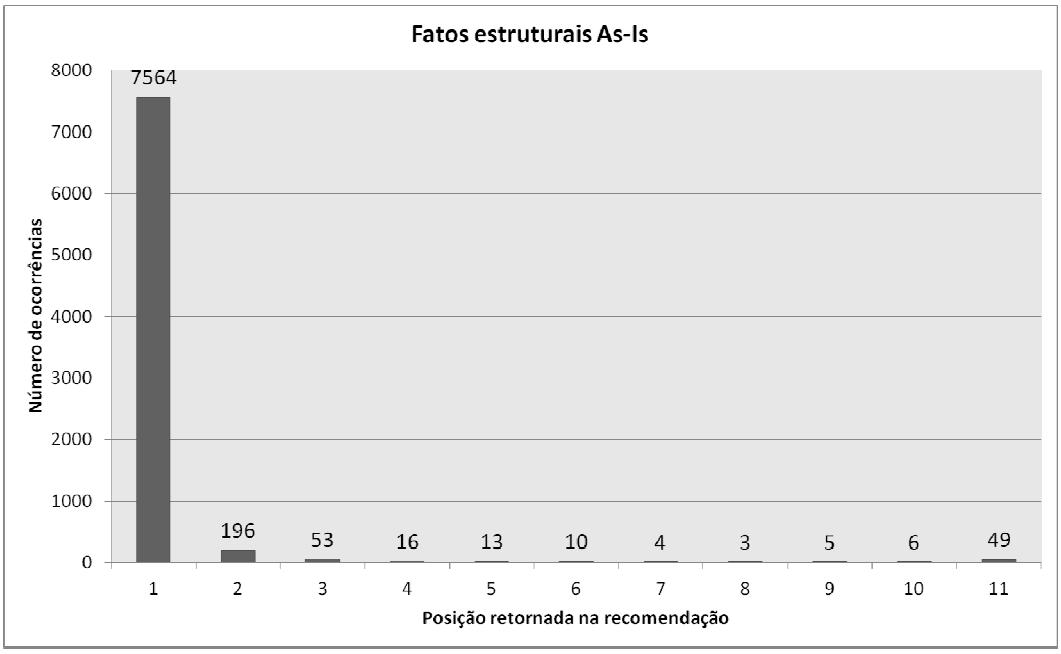

Figura 14 Resultado das heurísticas aplicadas aos fatos estruturais *As-Is*

O histograma da Figura 14 mostra que a distribuição de freqüências das posições recomendadas para os candidatos originais ficou concentrada nas primeiras posições: em 7564 casos, cerca de 95% do total das consultas, o candidato esperado foi retornado já na primeira posição. Em apenas 0,60% das consultas o candidato esperado não foi retornado dentre as dez primeiras colocações. A princípio, o resultado mostrado pelo histograma da Figura 14 parece bastante positivo, mas percebeu-se que na verdade ele era artificial. Ao usar os fatos estruturais de um fragmento de código, levam-se em consideração as suas propriedades privadas. Consideram-se aqui propriedades privadas classes, atributos e métodos que tenham escopo privado ao projeto do fragmento original. Ao usar os fatos estruturais de um fragmento de código que está armazenado no repositório para realizar uma consulta neste mesmo repositório, as propriedades privadas irão contribuir de forma injusta para a pontuação deste candidato, visto que apenas os candidatos deste projeto podem dispor destes fatos estruturais. Em um cenário real de uso do sistema de recomendações, o fragmento de código em que o desenvolvedor está trabalhando não teria acesso a estas informações privadas contidas nos exemplos do repositório. Por este motivo, uma segunda avaliação foi realizada a fim de amenizar o ruído introduzido pelas propriedades privadas. Desta forma buscou-se obter um melhor indicador sobre a suficiência do modelo de dados.

Numa segunda avaliação, o experimento base da Seção 5.1 foi modificado de modo que os fatos estruturais de um candidato fossem extraídos sem levar em conta as suas propriedades privadas. A fim de automatizar o processo de identificação destas, foram consideradas propriedades privadas quaisquer propriedades declaradas no contexto do projeto ao qual pertence o fragmento base. O contexto do projeto do fragmento original foi determinado a partir do prefixo que nomeia a sua estrutura de diretório. No caso de um método `br.pucrio.inf.util.Exemplo.foo`, por exemplo, qualquer propriedade cujo nome completamente qualificado tivesse o prefixo `br.pucrio.inf` seria considerada como propriedade privada e por isso removida dos fatos estruturais usados para realizar a consulta. Esta é uma estratégia demasiadamente cautelosa, a qual remove inclusive propriedades que têm visibilidade pública no escopo do projeto. Entretanto, esta estratégia não influenciará positivamente o resultado das heurísticas; pelo contrário, irá diminuir o conjunto de fatos estruturais disponíveis para as heurísticas. Desta forma é mantida sempre uma avaliação pessimista das heurísticas, como de fato se deseja. A Figura 15 mostra o histograma que apresenta o resultado obtido ao aplicar as heurísticas nos candidatos após terem suas propriedades privadas removidas dos fatos estruturais.

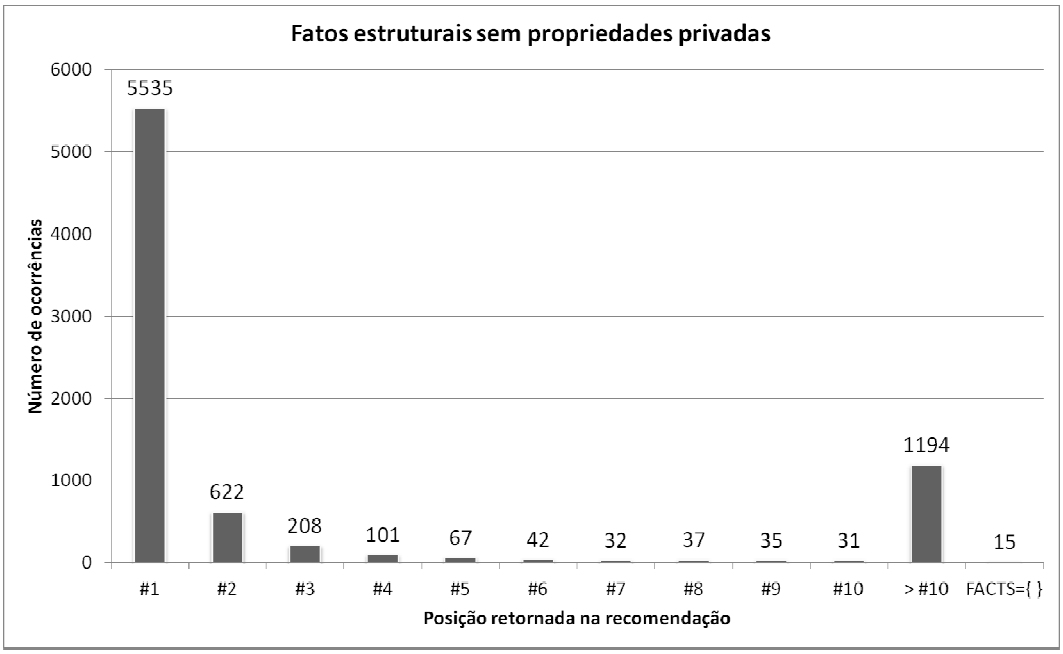

Figura 15 Resultado das heurísticas aplicadas aos fatos estruturais sem propriedades privadas

No histograma da Figura 15, cerca de 70% das consultas realizadas retornaram o candidato esperado na primeira posição e cerca de 85% das consultas retornaram o candidato esperado entre as dez primeiras posições. Ressalta-se assim a capacidade dos fatos estruturais propostos no modelo de dados em identificar fragmentos de código similares estruturalmente.

Vale salientar ainda que o número de consultas que não retornam o candidato esperado dentre os dez primeiros candidatos subiu consideravelmente em relação ao primeiro experimento: de 0,60% subiu para 15,0%. A principal razão para este aumento significativo no número foi a própria configuração do experimento. Ao remover as propriedades privadas de alguns candidatos seus conjuntos de fatos estruturais tornaram-se tão pequenos que as consultas não foram capazes de identificar no repositório o candidato esperado. Houve inclusive casos extremos em que após a remoção das propriedades privadas o conjunto de fatos estruturais tornou-se vazio. Estes casos são representados no histograma pela barra com o rótulo `FACTS = { }`. Todos os candidatos que continham apenas fatos estruturais relativos a propriedades privadas foram removidos do repositório de exemplos. Além disso, houve casos em que após a remoção das propriedades privadas o conjunto de fatos estruturais de alguns candidatos continha fatos excessivamente ordinários, como chamadas aos métodos `toString()`, `equals()`, `getClass()`, ou ainda o uso dos tipos `String`, `Double`, `Integer` e `Object`. Percebe-se assim que fatos excessivamente ordinários têm pouco poder de realçar as similaridades estruturais de fragmentos de código.

5.1.2. Avaliação do ajuste dos pesos

Como definida na Seção 3.4.2.2, a função de pontuação de candidato apresenta um fator de atribuição de peso aos termos que pode ser ajustado a cada sessão de utilização do sistema de recomendações. Podem-se definir pesos para cada campo distinto dos termos de uma consulta, i.e., podem ser atribuídos pesos diferentes para os termos: `HANDLES`, `CALLS` e `USES`.

Na seção anterior, os experimentos foram realizados usando os valores padrões para os pesos dos campos. O peso padrão tem valor igual a 1 para todos os campos. Nesta seção, será apresentado o experimento usado para verificar se o

ajuste dos pesos permite identificar com melhor precisão fragmentos de código estruturalmente similares. O objetivo do experimento não foi determinar uma configuração "ótima" com valores exatos para cada peso, mas sim tentar determinar para o repositório usado qual a melhor relação entre os pesos.

O experimento realizado foi configurado da seguinte maneira. Primeiramente, definiram-se diversas configurações de valores a serem usados como pesos na construção das consultas. Para tanto, foi definido um conjunto base de pesos: `BASE = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`. A partir deste conjunto base de pesos foram definidas todas as permutações (considerando repetição de elementos) de tamanho três: `{ {1,1,1}, {1,1,2}, {1,1,3} ... {10, 10, 10} }`. Ao todo, foram geradas mil permutações diferentes. Para cada permutação de pesos distinta, o experimento da Seção 5.1.1 era executado, considerando-se o primeiro elemento da permutação o peso para o campo `HANDLES`, o segundo elemento o peso para o campo `CALLS` e o terceiro elemento o peso para o campo `USES`. Optou-se por reusar o experimento da Seção 5.1.1 ao invés do experimento base da Seção 5.1, para evitar o ruído causado pelas propriedades privadas nos resultados recomendados e assim observar melhor a influência dos pesos nos resultados das recomendações. Por fim, definiram-se duas métricas para comparar as configurações de peso usadas. A primeira métrica contabiliza para uma determinada configuração de pesos quantos candidatos são recomendados já na primeira posição da lista de recomendação. Esta métrica é uma medida do melhor caso. Já a segunda métrica contabiliza para uma determinada configuração de pesos quantos candidatos não são recomendados entre as dez primeiras posições. Esta é uma medida do pior caso.

Visto que para cada permutação diferente e para cada candidato armazenado no repositório uma consulta era realizada, neste experimento foram realizadas ao todo 7.904.000 consultas ao repositório. É importante relembrar que o repositório de exemplos continha originalmente 7919 exemplos, mas 15 foram removidos, pois possuíam apenas propriedades privadas. Devido a este grande número de dados gerados, apenas um resumo destes dados serão apresentados na Tabela 3.

Tabela 3 Melhores e piores configurações de pesos

Métrica	Configuração	
	Melhor	**Pior**
Encontrar candidato original na 1ª posição	HANDLES=10, CALLS=3, USES=10	HANDLES=1, CALLS=10, USES=1
Não encontrar o candidato original entre as 10 primeiras posições	HANDLES=10, CALLS=5, USES=10 e HANDLES=9, CALLS=5, USES=10	HANDLES=1, CALLS=1, USES=1

A Tabela 3 resume os dados das configurações de pesos que tiveram os melhores e piores resultados para as duas métricas. Para fins de comparação, a configuração padrão usada no experimento da Seção 5.1.1, a qual atribui valor 1 para todos os pesos, obteve os seguintes resultados:

- Métrica I: 5535 casos em que o candidato original era retornado na 1ª posição
- Métrica II: 1194 casos em que o candidato original não era retornado entre as 10 primeiras posições.

Para a primeira métrica, a configuração `{10, 3, 10}` foi a que obteve o melhor resultado, sendo capaz de encontrar o candidato original já na primeira posição em 5725 vezes, das 7904 tentativas. Já a configuração `{1, 10, 1}` foi a de pior resultado, encontrando o candidato original já na primeira posição 5441 vezes.

Para a segunda métrica, as configurações `{10, 5, 10}` e `{9, 5, 10}` empataram com o melhor resultado: deixaram de encontrar o candidato original entre as 10 primeiras recomendações em 955 casos, do total de 7904 casos. Já a pior configuração foi a configuração padrão, `{1, 1, 1}`, a qual não foi capaz de encontrar o candidato original entre as 10 primeiras posições em 1194 dos casos.

Aparentemente, o número de candidatos das melhores configurações de pesos não são tão distantes dos valores alcançados pela configuração padrão. Entretanto, esta avaliação baseada apenas nos valores absolutos alcançados é superficial e ao não se considerar a dispersão da amostra coletada as conclusões tiradas podem ser um tanto quanto falaciosas. A Tabela 4 mostra a posição ocupada pela configuração padrão para cada um dos critérios.

Tabela 4 Posições da configuração padrão de pesos

Configuração	Posição de acordo com a Métrica I	Posição de acordo com a Métrica II
[1, 1, 1]	983	1000

De acordo com primeira métrica, a configuração padrão conseguiu retornar na primeira posição 5535 dos 7904 casos. Este resultado corresponde a 983ª melhor posição, dentre 1000. Para a segunda métrica, a configuração padrão ficou na 1000ª melhor posição, ou seja, foi a pior configuração possível. Logo, percebe-se que o ajuste dos pesos dos termos das consultas conseguiu melhorar significativamente os resultados obtidos pelas heurísticas quando comparados aos resultados obtidos pela configuração padrão. Apesar de ser difícil afirmar qual o valor "ótimo" para os pesos, é possível ao menos perceber um padrão na relação entre os valores dos pesos, como mostra a Tabela 5.

Tabela 5 10 melhores e 10 piores configurações

10 melhores configurações de acordo com a Métrica I	10 melhores configurações de acordo com a Métrica II
[10, 3, 10]	[10, 5, 10]
[10, 3, 9]	[9, 5, 10]
[9, 3, 10]	[8, 5, 10]
[8, 3, 10]	[9, 6, 10]
[9, 3, 9]	[10, 5, 9]
[10, 3, 7]	[9, 5, 9]
[10, 2, 10]	[10, 6, 10]
[10, 5, 10]	[8, 6, 10]
[10, 3, 8]	[10, 7, 10]
[10, 2, 7]	[7, 5, 10]
10 piores configurações de acordo com a Métrica I	**10 piores configurações de acordo com a Métrica II**
[1, 1, 1]	[1, 10, 1]
[1, 10, 1]	[1, 9, 1]
[1, 9, 1]	[1, 8, 1]
[1, 8, 1]	[1, 1, 1]
[1, 7, 1]	[1, 7, 1]
[1, 6, 1]	[1, 6, 1]
[1, 5, 1]	[1, 5, 1]
[1, 2, 1]	[1, 2, 1]
[1, 4, 1]	[1, 4, 1]
[1, 3, 1]	[1, 3, 1]

A Tabela 5 mostra as 10 configurações que obtiveram os melhores e piores resultados para cada uma das métricas. É possível perceber que não houve uma configuração que obteve o melhor resultado em todas as métricas, nem uma configuração que obteve o pior resultado em todas as métricas. Entretanto, é perceptível que em todas as melhores configurações um padrão se repete: o peso usado para o campo `CALLS` (o elemento do meio) é sempre inferior ao peso usado para os campos `HANDLES` (elemento da esquerda) e `USES` (elemento da direita).

O oposto ocorre para as piores configurações: o peso usado para o campo `CALLS` é sempre superior ao peso usado para os campos `HANDLES` e `USES`. Estes dados sugerem que para o repositório de exemplos usado, a melhor configuração de pesos a ser usada deve ter uma relação do tipo: `CALLS` < `USES` ≈ `HANDLES`.

Esta relação entre os pesos foi um tanto quanto surpreendente. A princípio, esperava-se que fatos estruturais mais raros no repositório tivessem maior potencial de realçar similaridades entre fragmentos de código. Por isso, esperava-se que a relação entre os pesos fosse da forma: `USES` < `CALLS` < `HANDLES`, pois achava-se que os tipos de variáveis fossem os fatos estruturais mais comuns no repositório, enquanto os tipos de exceção fossem os fatos mais raros no repositório. O que ocorreu na prática foi que muitos exemplos do repositório invocavam métodos ordinários, tais como `Object.toString()`, `String.equals()`, `String.length()`, dentre outros. Desta forma, ao se atribuir um peso maior às invocações de métodos, os fatos estruturais ordinários acabavam por realçar similaridades com outros fragmentos de código que também os possuíam, não sendo necessariamente o fragmento original. Isso acabou fazendo com que as heurísticas não fossem capazes de identificar o fragmento de código original, confundindo-o com outro fragmento que também continha os mesmos fatos ordinários.

Este mesmo fenômeno também ocorreu com os fatos estruturais relacionados aos tipos das variáveis usadas. Entretanto, o fenômeno com os fatos estruturais relacionados aos tipos das variáveis usadas foi ofuscado pelo tipo da exceção tratada. Visto que o tipo da exceção tratada é um parâmetro passado para o bloco `catch`, este tipo também foi considerado como um fato estrutural relacionado ao tipo da variável usada. Desta forma, ao se atribuir um alto peso à heurística do tipo da variável usada, atribuía-se indiretamente também um alto peso à heurística do tipo da exceção tratada.

Perceba na

Tabela 5 que, considerando as melhores configurações, valores altos para o peso da heurística do tipo da exceção tratada (elemento mais à esquerda) são sempre acompanhados por valores altos para o peso da heurística do tipo da variável usada (elemento mais à direita). Na prática, as melhores configurações obtiveram os melhores resultados devido ao maior peso atribuído ao tipo da exceção tratada. Isto se deve ao fato de que os tipos das exceções tratadas são fatos estruturais mais raros no repositório e, portanto, com maior potencial de realçar as similaridades entre dois fragmentos de código. Ou seja, a crença inicial de que fatos estruturais mais raros no repositório têm maior potencial de realçar similaridades entre fragmentos sustentou-se no fim das contas. Apenas a relação entre os pesos que não se confirmou. Além disso, percebeu-se que: (i) os fatos estruturais ordinários, não só têm pouco potencial de realçar similaridades estruturais entre fragmentos de código (como já havia sido afirmado na Seção 5.1.1), como podem até atrapalhar a precisão das heurísticas em alguns casos; e (ii) os fatos estruturais relacionados ao tipo da exceção possuem bom potencial para realçar similaridades entre fragmentos de código.

5.2. Avaliação de relevância

Finalizada a implementação do sistema, outras avaliações mais detalhadas foram realizadas. O principal serviço oferecido pelo sistema de recomendações é o provimento de fragmentos de código que sejam relevantes aos desenvolvedores em tarefas de implementação de código de tratamento de exceções. Portanto, fez-se necessário avaliar a relevância dos fragmentos de código que são recomendados pelo sistema de recomendações. As avaliações de relevância das recomendações tiveram caráter exploratório-qualitativo e objetivaram verificar as seguintes questões:

Q3. As heurísticas são capazes de recomendar informações relevantes para a implementação do tratamento de exceções?

Q4. O tamanho do conjunto de fatos estruturais e a proveniência dos fatos estruturais usados pelas heurísticas influencia na relevância das recomendações na atividade de implementação do tratamento de exceções?

A questão Q3 motivou a realização de uma avaliação de natureza mais qualitativa a fim de se verificar se as heurísticas de fato recomenda informações relevantes para a implementação do tratamento de exceções como elas se propõem. Nesta avaliação, vinte fragmentos de código foram extraídos de duas aplicações baseadas no *framework* Eclipse para servirem de entrada para as heurísticas. Para cada fragmento, as heurísticas foram executadas e as dez primeiras recomendações foram categorizadas em "Relevantes" ou "Irrelevantes".

O sistema de recomendações foi modificado de modo que as dez primeiras recomendações fossem salvas em arquivos de texto para que a categorização de relevância fosse feita manualmente com base em um oráculo. O oráculo foi construído através de uma compilação de informações a respeito de operações usadas para o tratamento de exceções no contexto de aplicações baseadas no *framework* Eclipse. Além disso, o avaliador das recomendações já possuía experiência prévia em análises de qualidade de código de tratamento de exceções adquirida em trabalhos realizados em colaboração com pesquisadores da Universidade Federal do Rio Grande do Norte – UFRN – e também em estudos preliminares realizados no contexto desta dissertação (Seção 2.3). Mais detalhes a respeito desta avaliação e dos resultados obtidos serão discutidos na Seção 6.2.1.

A questão Q4 motivou uma análise mais profunda dos dados coletados na primeira avaliação. Buscou-se com esta análise avaliar qual a influência do tamanho do conjunto de fatos estruturais e também qual a influência da proveniência dos fatos estruturais na relevância das recomendações. Por proveniência dos fatos estruturais diz-se se os fatos estruturais estão relacionados a elementos do *framework*, da aplicação ou de APIs de terceiros. Dado que os fatos estruturais usados pelas heurísticas são extraídos do fragmento de código em que o desenvolvedor está trabalhando, buscou-se assim ter um indicador indireto de o quão completo deve ser o fragmento de código que o desenvolvedor está trabalhando para que as heurísticas consigam fazer recomendações relevantes. Desta forma, nesta análise cruzaram-se os dados a respeito da relevância dos exemplos recomendados com o tamanho do conjunto de fatos estruturais usados pelas heurísticas e com a proveniência dos fatos estruturais. Mais detalhes a respeito desta análise e dos resultados obtidos serão discutidos na Seção 6.2.2.

Finalmente, destaca-se que os resultados obtidos nas avaliações realizadas estão fortemente ligados à amostra utilizada para popular o repositório de

exemplos e por isso não podem ser generalizados para outros contextos. Porém, visto que o domínio das aplicações escolhidas para a amostra usada na avaliação conta com uma diversidade razoável de características, como aplicações de propósito e tamanho bastante distintos, heterogeneidade de tipos de exceções, tratadores de naturezas distintas, dentre outras, e dado que a definição das heurísticas foi feita de modo independente de domínio, os resultados iniciais obtidos são bons indícios de que podem ser replicados em outras amostras. A finalidade principal das avaliações realizadas foi investigar se as contribuições esperadas das heurísticas de recomendações de exemplos de código para tratamento de exceções se concretizam em um subconjunto de aplicações de um domínio específico. Ademais, estas avaliações apontam para novos direcionamentos de pesquisa a serem tratados em trabalhos futuros.

5.2.1. Relevância dos exemplos recomendados

Todos os *frameworks* definem políticas de uso normal que, geralmente, são documentadas em APIs, tutoriais, FAQs, exemplos de código, etc. Alguns *frameworks* possuem também políticas próprias para o tratamento de exceções. O *framework* Eclipse, por exemplo, define um conjunto de diretrizes para operações básicas usadas em alguns tratadores de exceções (Eclipse-Wiki):

> A partir do Eclipse 3.3, foram incorporadas novas funcionalidades para tornar as tarefas de *logging* e de notificação ao usuário consistente em toda a plataforma Eclipse. Estas novas funcionalidades também permitem aos desenvolvedores de aplicações acoplar suas próprias ferramentas de diagnóstico a fim de fornecer uma melhor experiência ao usuário

Mesmo que sejam realizadas de maneira trivial em muitas aplicações convencionais, as tarefas de *logging* e de notificação de usuários podem tornar-se bastante complexas quando se trabalha no contexto de uma aplicação baseada em um *framework* complexo, como o Eclipse. O trecho de código abaixo exemplifica a simples operação de registrar em arquivo de *log* uma mensagem de erro no registro de erros de uma aplicação baseada no Eclipse:

```
try{
...
} catch( IOException e ){
    IStatus ioErrorStatus = new Status(
        IStatus.ERROR,
        MyPlugin.getIdentifier(),
        NLS.bind( Messages.INDEX_CORRUPT_MSG ),
        e );
    StatusManager.getManager().handle( ioErrorStatus,
           StatusManager.LOG );
}
```

Perceba que apenas para registrar no arquivo de *log* uma mensagem de erro é necessário usar 5 classes concretas e invocar 3 métodos, sendo 2 deles encadeados, além de identificar que os parâmetros do tipo inteiro do construtor de `Status` e do método `handle` são definidos pelas constantes `IStatus.ERROR` e `StatusManager.LOG`, respectivamente. Ou seja, para um desenvolvedor com pouca experiência no desenvolvimento de aplicações baseadas no *framework* Eclipse até mesmo uma tarefa trivial como registrar uma mensagem de erro em um arquivo de *log* pode ser bastante trabalhosa. Algumas destas informações até são documentadas pela *Eclipse Foundation Open Source Community,* mas geralmente estão espalhadas em diferentes documentos de diferentes naturezas: wikis, FAQs, documentação de APIs, etc. Uma possível fonte alternativa deste tipo de informações é o próprio código fonte de aplicações baseadas em um determinado *framework*. Mas dadas restrições típicas em prazos de entrega de projetos, o tempo necessário para a busca e compreensão *ad hoc* destas informações pode tornar-se impeditivo em muitos casos, independente da experiência do desenvolvedor.

Neste contexto, foi realizada uma avaliação a fim de determinar se as heurísticas usadas pelo sistema de recomendação são capazes de encontrar fragmentos de código contendo informações relevantes para o desenvolvedor quando estão implementado o tratamento de exceções em suas aplicações. Visto que a avaliação foi realizada pelo mesmos desenvolvedor e projetista do sistema sob análise, algumas medidas foram tomadas para tornar a avaliação menos tendenciosa. Primeiramente, definiu-se um oráculo para decidir quais informações seriam contabilizadas como relevantes ou não. Para isso, foi compilado um conjunto de diretrizes relacionadas à política de tratamento de exceções empregadas por aplicações baseadas no Eclipse. Tais diretrizes foram compiladas

de documentos diversos disponibilizados pela *Eclipse Foundation Open Source Community:* tutoriais, APIs, FAQs, fóruns de discussão e listas de e-mails. As diretrizes foram definidas em termos de classes e métodos que deveriam ser usados dentro dos blocos catch a fim de realizar tarefas como *logging*, notificação ao usuário, notificação aos controladores do *framework*, etc.

Após a definição do oráculo, definiu-se uma métrica binária para categorizar os candidatos recomendado em relevantes ou não relevantes: caso um candidato recomendado possua ao menos uma diretriz definida pelo oráculo, o candidato será considerado relevante e receberá nota 1; do contrário, será considerado irrelevante e receberá nota 0. Priorizou-se uma métrica binária a fim de facilitar a análise de relevância. Em trabalhos futuros pretende-se realizar experimentos controlados com desenvolvedores em que estes poderão pontuar a relevância das informações recomendadas em uma escala mais graduada.

Definidos o oráculo e a métrica usada para a avaliação, finalmente definiu-se o experimento a ser realizado. O experimento foi configurado da seguinte maneira. Inicialmente, foram selecionadas duas aplicações baseadas no Eclipse que obedeciam aos critérios definidos na Seção 4.2, mas que não foram utilizadas para popular o repositório. Dez métodos foram escolhidos aleatoriamente de cada aplicação, devendo obedecer somente ao critério de possuir mais de 15 linhas de código. Após escolhidos, os dez métodos foram rapidamente inspecionados apenas para se descartar qualquer caso de métodos clones ou métodos muito semelhantes. Em seguida, as heurísticas foram aplicadas a cada um dos métodos selecionados e os dez primeiros candidatos recomendados para cada método foram categorizados de acordo com a métrica de relevância definida. A Tabela 6 apresenta os resultados obtidos.

Tabela 6 Avaliação de relevância

	Rec. 1	Rec. 2	Rec. 3	Rec. 4	Rec. 5	Rec. 6	Rec. 7	Rec. 8	Rec. 9	Rec. 10
Candidato 1	0	1	1	1	1	1	1	1	1	1
Candidato 2	0	0	0	0	0	0	1	0	1	1
Candidato 3	0	0	0	1	1	1	1	0	0	0
Candidato 4	1	1	1	0	0	0	0	0	0	0
Candidato 5	1	1	1	0	1	1	1	1	0	0
Candidato 6	1	0	0	0	0	1	0	0	1	0
Candidato 7	0	1	1	1	1	1	0	1	1	1
Candidato 8	1	0	1	0	0	1	0	0	0	0
Candidato 9	1	1	0	0	0	0	0	1	1	1
Candidato 10	1	1	1	1	1	1	1	1	1	1
Candidato 11	1	1	1	1	0	0	0	0	1	0
Candidato 12	0	0	1	1	0	1	0	1	0	0
Candidato 13	1	1	1	1	1	1	0	0	0	0
Candidato 14	0	0	1	1	1	0	1	1	1	0
Candidato 15	1	0	0	1	0	1	0	0	1	0
Candidato 16	1	1	1	1	1	1	1	1	1	0
Candidato 17	0	0	1	1	1	0	1	1	0	0
Candidato 18	1	1	1	0	1	1	1	1	1	1
Candidato 19	1	0	1	0	0	0	1	0	0	0
Candidato 20	0	0	0	1	1	1	1	1	1	1

A Tabela 6 mostra as notas de relevância das recomendações de cada candidato. Para todos os vinte candidatos usados foi encontrada ao menos uma recomendação com informações relevantes dentre as dez primeiras recomendações retornadas. Percebe-se assim que as heurísticas usadas para recomendação de exemplos de código de tratamento de exceções permitem encontrar informações relevantes ao desenvolvedor.

Dos vinte métodos usados como base para as buscas, apenas o Candidato 2 não recebeu recomendações relevantes dentre as cinco primeiras recomendações realizadas pelo sistema de recomendações. Ao analisar com mais detalhes estas cinco primeiras recomendações, percebeu-se um fato interessante. Todas as cinco primeiras recomendações retornadas invocavam o `método IResourceTree.failed( IStatus )` em seus blocos catch da seguinte maneira:

```
try{
...
} catch (CoreException e ){
    resourceTree.failed( e.getStatus() );
}
```

Ao verificar a documentação do método `IResourceTree.failed( IStatus )` confirmou-se que se tratava de um método específico para o tratamento de exceções associadas ao tipo `IResourceTree`:

`void IResourceTree.failed( IStatus reason )` – declara que a operação subjacente falhou pela razão especificada por `reason`

Como este tratamento não havia sido identificado inicialmente na compilação do oráculo e para que os critérios não fossem modificados durante a avaliação, definiram-se as notas das cinco primeiras recomendações do Candidato 2 como sendo igual a 0. Entretanto, pode-se considerar que estas recomendações continham também informações relevantes. Na verdade, este fato realça a concretização da principal característica e idéia motivadora do sistema de recomendação para código de tratamento de exceções: a capacidade de prover informações ao desenvolvedor de modo a ajudá-lo no processo de aprendizado de como tratar exceções no contexto de sua aplicação. As informações contidas nas recomendações foram o ponto de partida para a aquisição de um novo conhecimento.

Como mencionado anteriormente, as heurísticas foram capazes de recomendar informações relevantes para todos os candidatos usados nas avaliações. Foram mínimos os casos em que o uso do sistema não recomendou informações relevantes, o que certamente aumenta a confiança dos desenvolvedores-usuários no sistema. Entretanto, foi verificado em muitos casos que os exemplos de código recomendados continham informações relevantes, mas eram métodos muito grandes. Alguns destes métodos continham mais de 300 linhas de código, com muitas outras informações irrelevantes para o usuário. Estes cenários apontam para três melhorias imediatas a serem possivelmente implementadas futuramente no sistema de recomendação a fim de contornar este problema.

A primeira melhoria é implementar uma melhor apresentação para os candidatos recomendados. Ao invés de apresentar o código completo dos candidatos recomendados, uma melhor apresentação poderia conter apenas as informações que são alcançáveis pelas informações usadas dentro do bloco `catch` do candidato.

A segunda melhoria em vista é considerar na função de pontuação o tamanho dos candidatos recomendados, de modo que a função de pontuação tenha um fator proporcional ao inverso do tamanho dos candidatos. Ou seja, quanto maior um candidato, menor será a sua nota. O tamanho dos candidatos pode ser inferido pelo número de linhas de código, ou ainda pelo número de estruturas sintáticas. Tais informações podem ser facilmente recuperadas com pequenos ajustes no analisador sintáticos implementado no mecanismo de coleta de informações.

Já a terceira melhoria consiste em aprimorar o processo de coleta de informações. Ao melhorar a qualidade das informações armazenadas no repositório, conseqüentemente melhorar-se-á a qualidade das recomendações realizadas. Dado que em alguns casos os exemplos recomendados eram muito extensos, o aprimoramento do mecanismo de coleta de informações deverá passar a desconsiderar métodos muito grandes e não armazená-los no repositório de exemplos. Isto pode ser facilmente aprimorado usando técnicas para detecção da anomalia estrutural *Long Method* (FOWLER e BECK, 1999).

Apesar dos esforços do mecanismo de coleta de informações para não guardar no repositório candidatos implementado tratadores ineficazes, ainda ocorreram alguns casos em que a recomendação retornada para o desenvolvedor implementava um tratador ineficaz. A maioria destes casos não foi detectada originalmente pelo mecanismo de coleta de dados por serem variações ou combinações dos tratadores ineficazes detectados isoladamente, como por exemplo o tratador abaixo, um típico caso de `printStackTrace` "disfarçado":

```
try{
...
} catch (Exception e ){
    e.printStackTrace();
    return;
}
```

O tratador acima não traz nenhum tipo de informação relevante ao desenvolvedor. Entretanto, a combinação das expressões `printStackTrace` e `return` acabou "disfarçando" o tratador ineficaz e, por isso, passou despercebido pelo coletor de informações e foi armazenado no repositório de exemplos. Com ajustes mínimos na implementação do analisador sintático do mecanismo de coleta de dados estes tratadores ineficazes "disfarçados" serão desconsiderados.

5.2.2.
Influência do tamanho do conjunto de fatos estruturais

A fim de localizar exemplos de código relevante ao usuário, é necessário que a consulta construída pelo mecanismo de consultas contenha um ou mais fatos estruturais que sejam capazes de realçar potenciais bons candidatos. Os fatos estruturais usados pela consulta são extraídos do fragmento de código em que o desenvolvedor trabalha. Desta forma, o fragmento de código do desenvolvedor é quem deve, a priori, conter os fatos estruturais capazes de realçar potenciais bons candidatos a recomendação. Por este motivo, a segunda avaliação de relevância realizada nesta dissertação teve como objetivo investigar qual o impacto do tamanho e do conteúdo do fragmento de código do desenvolvedor nos resultados recomendados pelas heurísticas. Esta segunda avaliação de relevância analisou mais a fundo os dados coletados na avaliação da Seção 5.2.1, cruzando os dados sobre a relevância das recomendações realizadas com os dados relativos aos conjuntos de fatos estruturais que foram usados para realizar as buscas. A Tabela 7 mostra os dados usados nesta 2a avaliação.

Tabela 7 Avaliação de tamanho e proveniência dos fatos estruturais

Candidatos	Total de recomendações relevantes	Fatos estruturais			
		Eclipse	Java	Primitivos	Outros
Candidato 1	9	5	3	0	0
Candidato 2	3	8	3	2	0
Candidato 3	4	4	16	2	0
Candidato 4	3	7	7	1	0
Candidato 5	7	22	6	2	0
Candidato 6	3	25	8	1	0
Candidato 7	8	25	10	2	0
Candidato 8	3	15	3	1	0
Candidato 9	5	4	7	0	0
Candidato 10	10	17	4	2	0
Candidato 11	5	13	14	2	0
Candidato 12	4	23	4	2	0
Candidato 13	6	31	10	2	1
Candidato 14	6	16	7	3	0
Candidato 15	4	17	8	1	2
Candidato 16	9	20	5	1	0
Candidato 17	5	1	4	0	1
Candidato 18	9	0	2	0	4
Candidato 19	3	7	6	2	1
Candidato 20	7	30	7	1	0

A Tabela 7 mostra na primeira coluna o total de recomendações relevantes que cada candidato recebeu no experimento realizado na avaliação da Seção 5.2.1. Nas outras colunas, os fatos estruturais de cada candidato estão divididos em quatro grupos: (i) Eclipse, para os fatos estruturais que estão relacionados com tipos definidos pelo framework Eclipse; (ii) Java, para os fatos estruturais que estão relacionados com os tipos definidos pela linguagem Java; (iii) Primitivos, para os fatos estruturais que estão relacionados com os tipos primitivos definidos pela linguagem Java; e (iv) Outros, para os fatos estruturais que estão relacionados com tipos definidos em APIs de terceiros. O tamanho da amostra de dados não permite realizar afirmações categóricas a respeito de relações entre os dados analisados. Entretanto, analisando os dados da Tabela 7 foram observados cenários distintos que levaram a formulação de algumas hipóteses.

O Candidato 10, o qual recebeu todas as recomendações contendo informações relevantes, possui um conjunto de fatos estruturais de tamanho razoável: 23 elementos, sendo quase todos (17) fatos relacionados a tipos específicos do Eclipse. O candidato 16, que recebeu 9 recomendações relevantes, também continha um conjunto de fatos estruturais de tamanho razoável: 27

elementos, dos quais 20 estão relacionados a tipos do Eclipse. Formulou-se então a hipótese de que um maior número de fatos estruturais relacionados a tipos do *framework* influenciaria positivamente a relevância das recomendações. Entretanto, ao analisar o Candidato 6, o qual possui um conjunto de fatos estruturais contendo 35 elementos, dos quais 25 são relacionados a tipos do Eclipse, verificou-se que o número de recomendações relevantes foi bem pequena: apenas três. Aparentemente, uma contradição à hipótese formulada. Analisando em mais detalhes o conjunto de fatos estruturais do Candidato 6, mostrado abaixo, compreende-se melhor o porquê para tão poucas recomendações relevantes:

```
handles:java.io.IOException
handles:java.lang.Exception
calls:java.util.List<java.lang.String>.add
calls:org.eclipse.core.runtime.IProgressMonitor.beginTask
calls:org.eclipse.core.runtime.IProgressMonitor.done
calls:org.eclipse.core.runtime.IProgressMonitor.isCanceled
calls:org.eclipse.jgit.api.CloneCommand.call
calls:org.eclipse.jgit.api.CloneCommand.setBranch
calls:org.eclipse.jgit.api.CloneCommand.setBranchesToClone
calls:org.eclipse.jgit.api.CloneCommand.setCloneAllBranches
calls:org.eclipse.jgit.api.CloneCommand.setCredentialsProvider
calls:org.eclipse.jgit.api.CloneCommand.setDirectory
calls:org.eclipse.jgit.api.CloneCommand.setProgressMonitor
calls:org.eclipse.jgit.api.CloneCommand.setRemote
calls:org.eclipse.jgit.api.CloneCommand.setTimeout
calls:org.eclipse.jgit.api.CloneCommand.setURI
calls:org.eclipse.jgit.api.Git.cloneRepository
calls:org.eclipse.jgit.api.Git.getRepository
calls:org.eclipse.jgit.lib.Ref.getName
calls:org.eclipse.jgit.lib.Repository.close
calls:org.eclipse.jgit.transport.URIish.toString
calls:org.eclipse.jgit.util.FileUtils.delete
calls:org.eclipse.osgi.util.NLS.bind
uses:boolean
uses:java.lang.InterruptedException
uses:java.lang.String
uses:java.lang.reflect.InvocationTargetException
uses:java.util.ArrayList<java.lang.String>
uses:java.util.List<java.lang.String>
uses:org.eclipse.core.runtime.IProgressMonitor
uses:org.eclipse.core.runtime.NullProgressMonitor
uses:org.eclipse.jgit.api.CloneCommand
uses:org.eclipse.jgit.api.Git
uses:org.eclipse.jgit.lib.Repository
```

Dentre os 25 fatos relacionados a tipos do Eclipse, 19 estavam relacionados ao projeto `org.eclipse.jgit` (grifados no trecho de código acima). Como este projeto não foi adicionado ao repositório, nenhum candidato do repositório de exemplos possui fatos estruturais relacionados aos tipos privados dele. Ou seja, os 19 fatos estruturais relacionados ao projeto `org.eclipse.jgit` não auxiliaram em nada na busca de candidatos. Na prática, apenas 6 fatos estruturais relacionados ao Eclipse é que de fato tinham condições de ser úteis na busca.

Portanto, o comportamento apresentado pelo Candidato 6 não contradiz a hipótese de que fatos estruturais provenientes do *framework* têm maior chance de realçar bons candidatos se mantém, como levou a crer uma análise superficial dos dados coletados.

Já o Candidato 9, o qual possui um conjunto de fatos estruturais bastante reduzido, contendo apenas 8 elementos, recebeu nove de dez recomendações contendo informações relevantes. Abaixo estão apresentados os fatos estruturais do Candidato 9 que ajudam a explicar este fenômeno observado:

```
handles:java.lang.InterruptedException
handles:java.lang.reflect.InvocationTargetException
handles:java.lang.Exception
calls:java.lang.Throwable.getMessage
calls:java.lang.reflect.InvocationTargetException.getCause
calls:org.eclipse.jface.operation.IRunnableContext.run
calls:org.eclipse.jface.wizard.Wizard.getContainer
uses:java.lang.String
uses:java.lang.Throwable
uses:org.eclipse.jface.operation.IRunnableWithProgress
uses:org.eclipse.jface.wizard.IWizardContainer
```

O responsável pelas boas recomendações apresentadas para o Candidato 9 foi o fato estrutural `calls:org.eclipse.jface.operation.IRunnableContext.run`. Este fato estrutural é um fato raro no repositório de exemplos. Apenas 105 candidatos possuem este fato estrutural de um universo de 7904 candidatos, o que representa cerca de 1% do total. Este fato estrutural funcionou então como um bom filtro, ressaltando um subconjunto de candidatos com contexto bastante similar ao do Candidato 9. Este fenômeno ajudou a complementar a hipótese anterior. A relevância das informações recomendadas não está relacionada apenas com a proveniência dos fatos estruturais, mas também com a freqüência de um fato estrutural: fatos estruturais extraordinários, isto é, menos freqüentes no repositório de exemplos, tendem a realçar informações mais relevantes.

Um fenômeno curioso ocorreu com o Candidato 18. Com um conjunto de fatos estruturais contendo apenas 6 elementos, o sistema de recomendações foi capaz de recomendar nove candidatos relevantes. Abaixo estão os fatos estruturais do Candidato 18:

```
handles:org.antlr.runtime.RecognitionException
handles:java.lang.Exception
calls:org.antlr.runtime.BaseRecognizer.match
calls:org.antlr.runtime.BaseRecognizer.pushFollow
calls:org.antlr.runtime.BaseRecognizer.reportError
uses:java.lang.Object
```

Quatro dos fatos estruturais do Candidato 18 eram relacionados a tipos de uma API de terceiros (grifados no trecho de código acima). Para estes quatro fatos, não houve nenhuma correspondência no repositório de exemplos. Sobraram então apenas dois fatos estruturais relacionados a tipos da linguagem Java: `uses:java.lang.Object` e `handles:java.lang.Exception`. Mesmo com apenas estes dois fatos extremamente genéricos, ainda foram realizadas 9 recomendações relevantes. Admite-se, no entanto, que este foi um caso bastante particular e extremamente eventual, e não um caso de competência do sistema de recomendação.

5.3. Comparação com trabalhos relacionados

5.3.1. Tratamento de exceções

Os trabalhos relacionados a ferramentas e técnicas de auxílio ao desenvolvedor em atividades relacionadas ao tratamento de exceções em sistemas de software apresentadas na Seção 3.1.1 têm objetivos distintos do objetivo do sistema de recomendação para código de tratamento de exceções proposto nesta dissertação. Essas ferramentas e técnicas apresentadas têm como objetivo principal auxiliar os desenvolvedores em tarefas de manutenção e compreensão de código de tratamento de exceções. Todas elas se baseiam em ferramentas de análise estática que capturam informações relacionadas a tratadores e interfaces excepcionais ou informações relacionadas aos fluxos de propagação das exceções. Além disso, essas ferramentas e técnicas apresentam as informações coletadas de forma textual ou visual.

Já o sistema de recomendação para tratamento de exceções tem como objetivo auxiliar os desenvolvedores em atividades relacionadas a implementação de código de tratamento de exceções. Seja para implementar um novo tratador, seja para melhorar a implementação de um tratador já existente. A principal diferença entre o sistema de recomendação para código de tratamento de exceções e os trabalhos relacionados mostrados na Seção 3.1.1 está relacionado ao cenário de uso. As ferramentas e técnicas da Seção 3.1.1 se propõem a auxiliar desenvolvedores a compreender e detectar problemas em um código já

implementado de forma inapropriada. Já o sistema de recomendação propõe-se a auxiliar o desenvolvedor a construir código de tratamento de exceção bem estruturado desde o princípio. O sistema de recomendação também pode ser usado para auxiliar os desenvolvedores a melhorar a qualidade de um tratador existente. Neste cenário, as ferramentas e técnicas da Seção 3.1.1 poderiam ser usadas conjuntamente ao sistema de recomendação: elas detectariam tratadores que necessitam ser melhorados e o sistema de recomendação auxiliaria o desenvolvedor a melhorar este tratador. Entretanto, a configuração deste cenário hipotético, no qual se combina o uso do sistema de recomendação para código de tratamento de exceções com outras ferramentas de apoio, demandaria outros tipos de avaliações que fogem ao escopo desta dissertação.

Há em comum entre os trabalhos relacionados e o sistema de recomendação para código de tratamento de exceções o uso de mecanismos de análise estática para a extração de informações do código fonte. A análise estática realizada pelo sistema de recomendação é usada para a extração dos exemplos de código e dos fatos estruturais. Esta análise estática é bastante simples e, assim como a análise estática realizada pelo compilador Java (Seção 3.1), baseia-se nas interfaces excepcionais declaradas pelos desenvolvedores. Em contrapartida, as análises estáticas realizadas pelos trabalhos relacionados são bem mais robustas e capazes de extrair informações mais precisas. Em trabalhos futuros, pretende-se incorporar ao mecanismo de extração de fatos e exemplos do sistema de recomendação técnicas de análise estática mais robustas. Possivelmente, as técnicas propostas pelos trabalhos relacionados serão adaptadas e reusadas, a fim de coletar informações ainda mais precisas que implicarão em recomendações melhores.

5.3.2. Sistemas de recomendação

A Tabela 8 resume os trabalhos relacionados a sistema de recomendação apresentados na Seção 2.2 e o sistema de recomendação para código de tratamento de exceções proposto nesta dissertação, comparando-os em cinco dimensões: objetivo de uso, informações usadas, acionamento das recomendações, fonte de informações e método de avaliação.

Tabela 8 Comparação de trabalhos relacionados

Trabalho	Objetivo	Informações usadas	Fonte de informações	Avaliação
MICHAIL, 2000	Auxiliar desenvolvedores a compreender padrões de uso de frameworks e	Relações de herança entre classes, tipos dos objetos instanciados, chamadas de métodos, métodos sobrescritos	Código fonte de aplicações baseadas no KDE	Argumentação e exemplificação de cenários de uso
HILL e RIDEOUT, 2004	Auxiliar desenvolvedores a reusar código existente	Tipo retornado pelo método, métricas de complexidade ciclomática, e tokens definidos de acordo com a especificação da linguagem Java	Código fonte da aplicação jEdit e das bibliotecas JavaSwing e JavaLanguage	Experimento analítico seguido de análise qualitativas dos resultados
MANDELIN et al., 2005	Auxiliar desenvolvedores a usar frameworks e APIs	Tipo do objeto que se tem posse, tipo do objeto que se deseja obter e sequência encadeada de chamadas de métodos	Código fonte de aplicações baseadas no Eclipse	Experimento analítico seguido de análise qualitativa dos resultados e experimento com desenvolvedores
YE e FISCHER, 2005	Auxiliar desenvolvedores a reusar código existente	Comentários e assinatura de funções	Documentação da especificação da linguagem Java, documentação da API da	Experimento analítico seguido de análise qualitativa dos resultados e experimento com desenvolvedores
HOLMES, WALKER e MURPHY, 2006	Auxiliar desenvolvedores a usar frameworks e APIs	Assinatura do método, identificadores e tipos das variáveis usadas, super-tipo das variáveis usadas, métodos invocados	Código fonte de aplicações baseadas no Eclipse	Experimento analítico seguido de análise qualitativa dos resultados, experimento com desenvolvedores, comparação com ferramentas de busca textual, avaliação da ferramenta em
BRUCH, MEZINI e MONPERRUS, 2010	Auxiliar desenvolvedores a compreender padrões de uso de frameworks e	Relações de herança entre classes, chamadas de métodos, métodos sobrescritos e métodos invocados	Código fonte de aplicações baseadas no Eclipse	Comparação da documentação gerada pela ferramenta com a documentação original provida pelo framework
Dissertação	Auxiliar desenvolvedores a implementar código de tratamento de	Tipo da exceção tratada, métodos chamados e tipo das variáveis usadas	Código fonte de aplicações baseadas no Eclipse	Experimento analítico seguido de análise qualitativa

Os trabalhos relacionados a sistemas de recomendação mostrados na Seção 2.2 podem ser divididos em dois grandes grupos em relação ao objetivo de uso: aqueles que se propõem a auxiliar desenvolvedores a trabalhar com *frameworks* e APIs, seja através do provimento de exemplos de código ou documentações extra; e aqueles que se propõem a auxiliar o desenvolvedor a encontrar código reutilizável. Em termos de objetivo de uso, o sistema de recomendação para código de tratamento de exceções é mais parecido com os sistemas que auxiliam o uso de *frameworks* e APIs através do provimento de exemplos de código, visto que o objetivo principal destes trabalhos é auxiliar na compreensão dos sistemas e não prover código para reuso.

Em relação às informações usadas para realizar as recomendações, todos os trabalhos usam informações estruturais do código fonte. Apenas o trabalho de Ye

e Fischer (2005) usa informações adicionais contidas em comentários de funções. O sistema de recomendação para código de tratamento de exceções usa a informação relativa ao tipo da exceção tratada que os demais trabalhos não usam. Na verdade, esta informação pode ser encarada como uma especialização de informações como "tipo das variáveis usadas" exploradas nos trabalhos (MICHAIL, 2000; HOLMES, WALKER e MURPHY, 2006). Esta maior precisão na informação a respeito do tipo tratado permite às heurísticas do sistema de recomendação para código de tratamento de exceções definir um peso ajustável para este tipo de informação. Além disso, permite também distinguir casos em que um método trata um tipo de exceção, dos casos em que um método sinaliza um tipo de exceção ou declara um tipo de exceção em sua interface excepcional. Evita-se assim que falsos positivos sejam recomendados para os casos em que um determinado método usa um tipo de exceção, mas não a trata localmente.

Em relação ao acionamento das recomendações, o sistema de recomendação para código de tratamento de exceções aqui proposto e quase todos os sistemas de recomendação discutidos na Seção 2.2 acionam as recomendações manualmente. Considera-se acionamento manual quando o desenvolvedor é quem solicita ao sistema as recomendações. O sistema proposto por Ye e Fischer (2005) é o único, dentre os apresentados, em que as recomendações são acionadas por um evento interno ao sistema. Neste sistema, as recomendações são acionadas no momento em que o desenvolvedor começa a declarar um método. Ao adotar o acionamento manual no sistema de recomendações corre-se o risco de os usuários continuarem a não dar atenção ao tratamento de exceções e nunca usarem as recomendações. Pretende-se em versões futuras do sistema incorporar um mecanismo de acionamento automático das recomendações ao se detectar uma exceção checada não tratada no código do desenvolvedor. Certamente, avaliações de usabilidade comparando as duas abordagens de acionamento serão necessárias para decidir qual a que melhor se adequa às necessidades dos desenvolvedores.

Em termos de avaliação dos sistemas, apenas Michail (2010) não seguiu uma avaliação mais criteriosa do seu sistema. Holmes, Walker e Murphy (2006) realizaram a mais completa avaliação, que incluía de experimentos analíticos a avaliações de uso em cenário de desenvolvimento na indústria. O sistema de recomendação para código de tratamento de exceções foi avaliado através de experimentos analíticos seguidos de avaliações qualitativas. O viés destas

avaliações foi o fato de o próprio desenvolvedor do sistema ter sido o avaliador. A fim de contornar esta limitação, definiu-se um oráculo para a relevância das informações recomendadas. Esta primeira avaliação indica que as recomendações realizadas pelo sistema exibem informações relevantes ao desenvolvedor. Para se obter uma maior certeza disto, pretende-se no futuro realizar experimentos controlados com desenvolvedores simulando cenários reais. O objetivo seria checar se as recomendações de fato são apreciadas por desenvolvedores desempenhando tarefas reais e se as recomendações de fato ajudam a melhorar a qualidade do código de tratamento de exceções quando comparada com cenários em que não foram utilizadas as recomendações.