2
Referencial teórico e estudos preliminares

Sistemas de software atuais são cada vez maiores em tamanho, número de usuários e complexidade estrutural. Para que sejam dignos de confiança, tais sistemas devem ser capazes de prover seus serviços corretamente mesmo operando sob condições excepcionais. Mecanismos de tratamento de exceções (GOODENOUGH, 1975) provêem facilidades para a detecção e a sinalização de condições excepcionais. Tais mecanismos provêem ainda funcionalidades para estruturar ações que remedeiem a ocorrência dessas condições excepcionais. Entretanto, implementar o tratamento de exceções adequadamente não é uma tarefa fácil de ser realizada. Mesmo desenvolvedores experientes têm dificuldade em implementar código de tratamento de exceções de boa qualidade (SHAH *et al.*, 2008a; SHAH *et al.*, 2010). A fim de atenuar esta situação, estudos recentes têm proposto ferramentas de apoio ao desenvolvimento e manutenção de código de tratamento de exceções (ROBILLARD e MURPHY, 2003; FU e RYDER, 2007; GARCIA e CACHO, 2011; SHAH *et al.*, 2008b). Entretanto, estas ferramentas falham em auxiliar desenvolvedores a implementar código de tratamento de exceções.

Estudos realizados com desenvolvedores de software (SHAH *et al.*, 2008a; SHAH *et al*., 2010) apontam que estes profissionais costumam estudar códigos desenvolvidos por terceiros como uma forma de aprender novas formas de implementar tratamento de exceções. Com a popularização de projetos de código aberto disponíveis em repositórios públicos, esta prática torna-se cada vez mais comum. Entretanto, a quantidade de informações disponíveis é enorme. Para se ter uma idéia, o *SourceForge*, maior portal de projetos de código aberto da Internet, abriga mais de 100.000 projetos de software. Sem o suporte adequado, desenvolvedores podem perder-se em meio a essa grande quantidade de informações disponíveis e gastar um tempo desproporcional para encontrar informações relevantes. Sistemas de recomendação, em especial, podem auxiliar

esses desenvolvedores a encontrar informações que sejam relevantes em suas tarefas.

Neste contexto, este capítulo traz um breve referencial teórico acerca de conceitos básicos e trabalhos relacionados a tratamento de exceções (Seção 2.1) e sistemas de recomendação para a engenharia de software (Seção 2.2). Por fim, também são apresentados os estudos exploratórios a respeito da qualidade do código de tratamento de exceções realizados preliminarmente no contexto desta dissertação (Seção 2.3).

2.1. Tratamento de exceções

Um sistema de software consiste em conjuntos de módulos que cooperam entre si a fim de prover um serviço para o meio externo (LEE e ANDERSON, 1990).

Def.: *Módulo de software* é um elemento de software bem delimitado que encapsula uma série de funcionalidades oferecidas ao meio externo através de uma interface bem definida (LEE e ANDERSON, 1990).

Def.: *Interface* de um módulo descreve as funcionalidades providas por este (LEE e ANDERSON, 1990).

Def.: *Serviço* de um sistema é o conjunto de funcionalidades providas por este sistema e que são perceptíveis pelo usuário (LEE e ANDERSON, 1990).

O projeto de um sistema de software define tanto as interações entre os módulos internos ao sistema, quanto as interações entre estes e módulos externos ao sistema. As interações entre os módulos de um sistema definem o seu comportamento dinâmico. Tal comportamento pode ser caracterizado como uma série de estados internos que um sistema adquire durante sua execução.

Def.: *Estado interno* de um sistema é o conjunto de condições no qual um sistema de software se encontra em um determinado instante no tempo durante sua execução (LEE e ANDERSON, 1990).

Sob condições ideais de uso, um sistema de software sempre executará através de transições válidas, transitando de um estado inicial conhecido para um estado final esperado. Uma transição define um relacionamento entre dois estados, indicando que o estado inicial irá passar para o estado final mediante a ocorrência de algum evento. Na prática, um sistema de software pode realizar algumas transições errôneas causadas por algum módulo defeituoso, ou devido ao mau projeto do sistema. Transições errôneas levam um sistema de software a estados inesperados, ou indesejados, que podem porventura impedir que o sistema proveja corretamente os seus serviços. Quando ocorre uma transição entre um estado em que o serviço é provido corretamente para um estado em que o serviço é provido incorretamente, é dito que houve um defeito de serviço, ou simplesmente defeito.

Def.: *Defeito* é um evento que altera o estado interno de um sistema de software para um estado em que o serviço não corresponde ao esperado, ou desejado, conforme definido em alguma especificação (LEE e ANDERSON, 1990).

Def.: *Erro* é um estado interno de um sistema de software que possibilita a ocorrência de um defeito (LEE e ANDERSON, 1990).

Def.: *Falha* é uma causa hipotética de um erro. Falhas podem ser imperfeições ou irregularidades que ocorrem em módulos de hardware ou software (LEE e ANDERSON, 1990).

A execução de um módulo de software pode ser dividida em dois fluxos de execução: o fluxo de execução normal e o fluxo de execução excepcional. O fluxo de execução normal define o fluxo de execução realizado para prover o seu serviço corretamente; e o fluxo de execução excepcional define o fluxo de execução realizado a fim de tolerar a manifestação de falhas que venham a ser detectadas. Quando a manifestação de uma falha impede o prosseguimento do fluxo normal de execução diz-se que ocorreu uma exceção.

Def.: *Exceção* é um evento excepcional que indica um estado interno inconsistente, que não permite a continuação do fluxo normal de execução de um programa.

Def.: *Fluxo de execução excepcional* é o fluxo de execução realizado por um módulo a fim de recuperar o seu estado interno para um estado correto (GARCIA *et al.*, 2001).

Mecanismos de tratamento de exceções são modelos usados para estruturar o fluxo excepcional de um módulo de software através da detecção e sinalização da ocorrência de exceções (GOODENOUGH, 1975). Estes mecanismos também permitem que módulos de software tomem medidas corretivas que recuperem o sistema para um estado correto. O mecanismo de tratamento de exceções é responsável por transferir o fluxo de execução de um módulo do seu fluxo de execução normal para o fluxo excepcional quando uma exceção é sinalizada. Após a exceção ser tratada, o mecanismo de tratamento de exceções retoma o fluxo de execução normal do sistema.

Def.: *Tratar uma exceção* é o ato de tomar medidas corretivas a fim de recuperar o estado de um sistema após detectar-se a ocorrência de uma exceção (GARCIA *et al.*, 2001).

No contexto de linguagens de programação, as exceções são tipicamente divididas em exceções pré-definidas e exceções definidas pelo usuário (LANG e STEWART, 1998). Exceções pré-definidas são declaradas implicitamente e estão relacionadas a condições excepcionais detectadas pelo mecanismo de suporte à execução da linguagem, ou ainda por hardwares e sistemas operacionais subjacentes. Exceções definidas pelo usuário são exceções definidas e detectadas no nível da aplicação de software. Pode-se ainda dividir as exceções definidas pelo usuário em exceções da aplicação e exceções de APIs (ou exceções de terceiros) (LANG e STEWART, 1998). Exceções da aplicação são exceções definidas no contexto da aplicação cliente, enquanto exceções de APIs são exceções definidas no contexto de bibliotecas reutilizáveis que provêem serviços à aplicação cliente.

Em linguagens de programação, mecanismos de tratamento de exceções são implementados ou como parte constituinte da linguagem, com elementos sintáticos próprios, ou como funcionalidades integradas à linguagem através de bibliotecas (DREW e GOUGH, 1990). No contexto de linguagens de programação, um mecanismo de tratamento de exceções define:

- Como representar uma exceção;

- Como especificar regiões protegidas da ocorrência de uma exceção;
- Como definir um tratador de exceções;
- Como associar uma determinada exceção a um tratador;
- Como sinalizar a ocorrência de uma exceção; e
- Como declarar na interface de um módulo quais exceções ele pode sinalizar.

Def.: *Região protegida* da ocorrência de uma exceção é um trecho de código que possui associado a si um ou vários tratadores de exceções (GARCIA et al, 2001).

Def.: *Tratador de exceção* é um conjunto de ações tomadas por um módulo de software a fim de reagir a uma exceção (GARCIA et al, 2001).

Tratadores de exceção típicos envolvem ações como: armazenar informações em arquivos de *log*, notificar ao usuário a detecção da exceção, realizar reconfigurações no sistema, liberar recursos alocados, tentar recuperar o estado do sistema por retrocesso, re-sinalizar a exceção detectada com outro tipo de exceção, dentre muitas outras.

Além de explicitarem a impossibilidade de o fluxo de execução normal prosseguir, as exceções também servem como meio de comunicação entre o módulo que requisita o serviço e o módulo que provê o serviço. Geralmente os tratadores requerem informações extras a respeito de uma exceção, como o seu nome, descrição, localização em que ocorreu, severidade, dentre outras. Algumas destas informações extras são passadas explicitamente pelo módulo que sinaliza a exceção, enquanto outras implicitamente pelo mecanismo subjacente de tratamento de exceções (LANG e STEWART, 1998).

As características apresentadas anteriormente são comuns a todos os mecanismos de tratamento de exceções em linguagens de programação. A concretização destas características, no entanto, pode variar bastante de linguagem para linguagem (GARCIA et al, 2001). Dentre as diversas implementações existentes, no contexto desta dissertação será explorado o mecanismo de tratamento de exceções implementado na linguagem Java. Optou-se por explorar o mecanismo implementado em Java dada a grande adoção da linguagem tanto em ambiente acadêmico quanto na indústria. É sabido que existem debates a respeito

das decisões de projeto do mecanismo de tratamento de exceções implementado por Java (CABRAL e MARQUES, 2007; ECKELS, 2003). Entretanto, a discussão da adequação destas decisões foge ao escopo desta dissertação e, por isso, não será realizada.

O mecanismo de tratamento de exceções implementado por Java (ARNOLD, GOSLING e HOLMES, 2000) representa exceções como objetos. Todos os objetos que representam exceções em Java devem ser instâncias de classes que herdem direta ou indiretamente da classe `Throwable`. Os tipos de exceção são organizados hierarquicamente em árvores de tipos, em que (idealmente) tipos representando condições excepcionais similares estão localizadas proximamente na árvore de tipos. A Figura 3 apresenta a árvore de tipos básicos de exceção de Java.

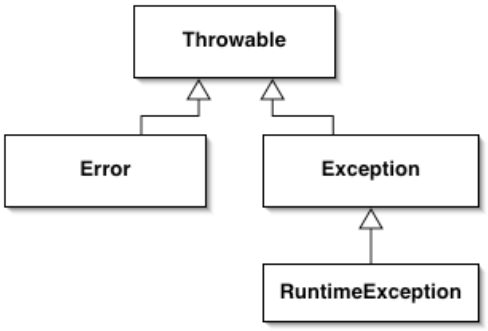

Figura 3 Hierarquia básica de tipos de exceção em Java

Como mostra a Figura 3, a classe `Throwable` possui duas subclasses bases diretas: `Error` e `Exception`. Exceções instâncias de `Error`, ou uma de suas subclasses, indicam problemas sérios que ocorrem externamente à aplicação. Aplicações em geral não têm como se recuperar de tais problemas, como por exemplo problemas na máquina virtual. A classe `Exception`, por sua vez, possui uma subclasse base direta, a classe `RuntimeException`. Exceções instâncias de `RuntimeException` representam condições excepcionais que são internas a aplicação. Estas condições geralmente são decorrentes de falhas de programação, como `NullPointerException` ou `ArrayIndexOutOfBoundsException`. Desta forma, exceções representadas por `RuntimeException`, e suas subclasses, representam condições excepcionais em que o código cliente não tem como recuperar-se adequadamente. Exceções instâncias da classe `Exception`, ou uma de suas

subclasses (que não sejam subclasse de `RuntimeException`), são geralmente usadas para representar condições excepcionais que aplicações bem estruturadas devem antecipar e se recuperar.

```
try
{
// Região protegida
}
catch ( E1 e ){  // Tratador para exceções instâncias do tipo E1 }
catch ( E2 e ){  // Tratador para exceções instâncias do tipo E2 }
```

O exemplo acima mostra a definição de regiões protegidas e tratadores em Java. Regiões protegidas são definidas através de blocos `try`. Os tratadores são definidos ao fim de um bloco `try` usando a cláusula `catch`. A associação entre uma determinada exceção e o seu respectivo tratador é feita através da definição de um parâmetro na cláusula `catch` que indica uma classe de exceção.

Uma exceção é tratada automaticamente através da invocação de um tratador adequado selecionado da lista de tratadores encontrados logo após o bloco `try`. Um tratador é considerado adequado caso a exceção a ser tratada seja instância da classe declarada na cláusula `catch`, ou ainda, instância de uma subclasse da classe declarada na cláusula `catch`. Quando um tratador captura uma exceção cujo tipo é subtipo da classe definida na cláusula `catch`, é dito que esta exceção foi capturada por subsunção[2] (ARNOLD, GOSLING e HOLMES, 2000).

As exceções em Java são elementos tipados da linguagem. Desta forma, é possível a realização de verificações estáticas a respeito do tratamento de algumas exceções. Para tanto, as exceções em Java são divididas em exceções checadas e exceções não-checadas.

[2] Do inglês *caught by subsumption*

Para as exceções checadas, um compilador para a linguagem Java verifica, em tempo de compilação, se existem tratadores associados a elas, analisando quais exceções checadas podem ser sinalizadas durante a execução de um método ou construtor. A ausência de um tratador para uma exceção checada ocasiona um erro de compilação. Um compilador para a linguagem Java ainda exige que exceções checadas sejam declaradas explicitamente na interface de métodos e construtores que podem sinalizar exceções desta natureza. Exceções checadas são representadas como instâncias da classe `Exception`, ou uma de suas subclasses (que não sejam subclasse de `RuntimeException`). As exceções não-checadas, por sua vez, não são verificadas pelo compilador e sua declaração na interface excepcional de métodos e construtores não é obrigatória. Exceções não-checadas são representadas como instâncias das classes `Error` ou `RuntimeException`, ou uma de suas subclasses.

Def.: *Exceções checadas* são exceções verificadas em tempo de compilação para que seja garantida a existência de tratadores associados às exceções desta natureza (ARNOLD, GOSLING e HOLMES, 2000).

Exceções são sinalizadas em Java através da cláusula `throw`. A interface excepcional de um método ou construtor indica quais exceções podem ser sinalizadas durante sua execução. Ela é declarada através da cláusula `throws` e deve indicar a classe, ou a superclasse, da exceção que pode ser sinalizada durante a execução do método ou construtor.

```
public File open ( String path ) throws IOException
{
    if( !isValid( path ) )
    {
          throw new MalformedPathException();
    }
    ...
}
```

No trecho de código acima, o método `open` recebe como parâmetro uma `String` e retorna um arquivo apontado pelo caminho indicado pelo parâmetro `path`. Antes de tentar abrir o arquivo, o conteúdo de `path` é validado para certificar-se de que o caminho indicado é válido. Caso o caminho indicado por `path` não seja válido, uma exceção do tipo `MalformedPathException` é sinalizada através da cláusula `throw`. Considerando que

`MalformedPathException` é uma exceção checada, faz-se necessário então ou declarar na interface excepcional do método a possibilidade de ocorrência desta exceção, ou tratá-la localmente com um bloco `try-catch`. No exemplo, optou-se pela primeira opção e a interface excepcional, especificada pela cláusula `throws`, indica que o método `open` pode sinalizar uma exceção do tipo `IOException`, a qual é superclasse de `MalformedPathException`.

Um compilador Java realiza suas análises estáticas para as exceções checadas baseado apenas na especificação das interfaces excepcionais definidas pelos desenvolvedores. Não é realizado nenhum tipo de análise estática mais aprofundada para garantir que um método que declara uma interface excepcional pode, de fato, sinalizar uma exceção.

```
public int sum( int x, int y ) throws Exception
{
    return x + y;
}

public void execute()
{
    sum(1, 2); //Erro de compilação: Unhandled exception type
Exception
}
```

No trecho de código acima, o método `sum` declara em sua interface excepcional a possibilidade de sinalizar uma exceção do tipo `Exception`, ainda que em seu corpo não haja nenhuma cláusula `throw` indicando a sinalização de uma exceção, nem a invocação de um método que possa sinalizar uma exceção. Ao invocar `sum` no corpo do método `execute` o compilador Java irá gerar um erro de compilação. O compilador irá acusar que é necessário tratar a exceção que pode ser sinalizada por `sum`, ainda que na prática não exista a possibilidade desta exceção ser sinalizada.

2.1.1. Trabalhos relacionados

2.1.1.1. Ferramentas de apoio ao desenvolvedor

Os primeiros trabalhos relativos a tratamento de exceções, propondo as primeiras notações (GOODENOUGH, 1975) e primeiras implementações em linguagens de programação (MACLAREN, 1977) foram concebidos há mais de

três décadas. Porém, ainda hoje se percebe que o tratamento de exceções é uma das atividades do desenvolvimento de software menos compreendidas por desenvolvedores em geral (ROBILLARD e MURPHY, 2003; SHAH *et al.*, 2008a; SHAH *et al.*, 2010). Diversos estudos apontam que falhas recorrentes e problemas de desempenho em sistemas de software estão relacionados com problemas no código de tratamento de exceções (BARBOSA e GARCIA, 2010; BARBOSA e GARCIA, 2011; CABRAL e MARQUES, 2007; COELHO *et al.*, 2008; SHAH *et al.*, 2008a; SHAH *et al.*, 2010; WEIMER e NECULA; 2004). Na última década, em especial, diversos trabalhos na literatura propuseram ferramentas que têm como objetivo apoiar o desenvolvedor em lidar com elementos de tratamento de exceções (CHANG *et al.*, 2001; CHANG *et al.*, 2002; FU e RYDER, 2007; ROBILLARD e MURPHY, 2003; SHAH *et al.*, 2008b). A seguir será apresentado um conjunto representativo destes trabalhos.

Chang et al. (2001) usam técnicas de análise estática para estimar os possíveis fluxos de propagação de exceções em aplicações Java independentemente das interfaces excepcionais declaradas pelos desenvolvedores. Estas técnicas de análise estática são mais refinadas do que as realizadas por um compilador Java convencional. Tais técnicas foram usadas para estimar: (i) tratadores desnecessários, isto é, blocos `try-catch` que rodeavam métodos que na prática não sinalizam exceções; e (ii) interfaces excepcionais desnecessárias, isto é, interfaces excepcionais declaradas em métodos que na prática não têm possibilidade de sinalizar exceções. Os autores mostram que para um conjunto de seis aplicações Java, um número significativo de tratadores e interfaces desnecessários foram detectados. Ademais, os autores argumentam que as técnicas propostas auxiliam desenvolvedores a simplificar o tratamento de exceções em suas aplicações ao remover do código fonte estruturas desnecessárias.

Posteriormente, Chang et al. (2002) também propuseram uma ferramenta de visualização para exibir os dados gerados por sua ferramenta de análise estática. A visualização provida apresenta textualmente para um determinado método selecionado quais exceções trafegam por ele, isto é, quais exceções o método selecionado não trata localmente e propaga adiante. Para cada exceção que trafega por um determinado método selecionado, também é exibido o local em que esta exceção foi levantada e o local final em que a exceção está sendo tratada.

Robillard e Murphy (2003) discutem as dificuldades encontradas por desenvolvedores ao implementar sistemas de software em linguagens com mecanismos de tratamento de exceções incorporados à linguagem. Em especial, discutem a dificuldade em se compreender os fluxos de propagação de uma exceção. Fluxos de propagação de exceção são os caminhos pelos quais uma exceção passa entre o local em que foi sinalizada e o local em que é tratada. Os fluxos de propagação de exceção são divididos em fluxos locais, em que uma exceção é sinalizada e tratada em um mesmo método, e fluxos globais, em que uma exceção é sinalizada e tratada em métodos diferentes. Os autores propõem um modelo para representar os fluxos de propagação de exceções e desenvolveram a ferramenta JEX para suportar este modelo em Java. Esta ferramenta extrai informações a respeito do fluxo de propagação de exceções e oferece visualizações em formato de grafos orientados para os diferentes tipos de exceções que podem trafegar em cada ponto da aplicação. Os autores utilizaram a ferramenta no código fonte de bibliotecas e aplicações Java e argumentam que a ferramenta JEX é útil principalmente no auxílio à compreensão dos fluxos globais.

Complementarmente ao trabalho de Robillard e Murphy (2003), Fu e Ryder (2007) propuseram a idéia de análise estática de exceções encadeadas. Uma exceção encadeada ocorre quando um tratador captura uma exceção e re-sinaliza esta exceção, possivelmente, mas não obrigatoriamente, com outro tipo de exceção. Quando uma exceção é capturada e re-sinalizada com outro tipo diz-se que esta exceção foi remapeada para outro tipo. Ao considerar as exceções encadeadas, os fluxos de propagação de exceções são analisados como longos fluxos em que possivelmente há mudança de tipo em alguns locais, ao invés de fluxos fragmentados, em que cada fragmento possui apenas um tipo de exceção. Este refinamento na análise de fluxos de propagação reduz consideravelmente o grande número de resultados apresentados pela ferramenta JEX, visto que o remapeamento de tipos de exceção foi bastante comum nas aplicações analisadas pelo estudo. Além disso, os autores argumentam que o grafo de propagação de exceções encadeadas representa melhor a arquitetura do código de recuperação de uma aplicação. Argumentam ainda que o grafo de propagação também é útil em tarefas de diagnóstico de falhas e compreensão do código de tratamento de exceções.

Shah et al. (2008b) propuseram a ferramenta de visualização ENHANCE – *ExceptioN HANdling CEntric visualization*. Tal ferramenta provê informações a respeito dos elementos do mecanismo de tratamento de exceção de Java através de três visualizações: a visão quantitativa, a visão de fluxo e a visão de contexto. A visão quantitativa exibe em forma de uma matriz o relacionamento entre elementos (pacotes, classes e métodos) que sinalizam e tratam exceções. A visão de fluxo representa através de um grafo o relacionamento entre métodos que lançam exceções, métodos que tratam exceções e os respectivos tipos de exceção. A visão de contexto estende a visão de fluxo exibindo em mais detalhes o fluxo de propagação das exceções. Além dos métodos em que se lançam e se tratam exceções, também são mostrados os métodos intermediários por onde a exceção trafega.

Todos os trabalhos apresentados nesta seção seguem uma mesma linha de pesquisa e podem ser vistos como refinamentos sucessivos de um mesmo paradigma: uma ferramenta de suporte à compreensão de fluxos de propagação cujo *back-end* é uma ferramenta de análise estática e o *front-end* apresenta gráfica ou textualmente os resultados gerados pela ferramenta de análise estática. A primeira limitação nestes trabalhos é que a quantidade de dados gerada pela ferramenta subjacente de análise estática é muito grande. Mesmo a ferramenta do trabalho de Fu e Ryder (2007), cujo objetivo é diminuir a quantidade de dados gerados em relação aos trabalhos de Chang et al. (2001) e de Robillard e Murphy (2003), apresentam dados na ordem de milhares. Esta enorme quantidade de dados na prática não auxilia os desenvolvedores na atividade de implementar código de tratamento de exceções. Além disso, parte destes dados é inútil aos desenvolvedores, pois pertencem a fluxos de propagação que, apesar de serem computados pela ferramenta de análise estática subjacente, em tempo de execução não existirão.

Outra limitação nos trabalhos apresentados anteriormente é que eles assumem que as aplicações analisadas apresentam um código de tratamento de exceções minimamente estruturados e que apenas melhorias pontuais são necessárias. Na prática, o que se percebe é que muitas aplicações e sistemas de software apresentam código de tratamento de exceções muito mal estruturados, para não se dizer quase inexistente. Todos os trabalhos descritos anteriormente adotam uma abordagem em que primeiro percebe-se que um sistema de software

apresenta código de tratamento de exceções mal estruturado e só em seguida medidas são tomadas a fim de melhorar este código. Mesmo que isso seja aplicado em projetos isolados, tal abordagem pode ser pouco eficaz. Dado o espalhamento do tratamento de exceções por diversos módulos, alterações no código de tratamento de exceções têm impacto global no sistema. Estas mudanças globais podem ser bastante onerosas e podem, inclusive, comprometer o projeto arquitetural do sistema (MACIA et al., 2012). Por fim, nenhum dos trabalhos descritos anteriormente apresenta uma alternativa ao desenvolvedor no sentido de implementar seu código de tratamento de exceções desde o princípio.

2.1.1.2. Boas práticas de desenvolvimento

Desde a publicação da primeira versão de *Design Patterns: Elements of Reusable Object-Oriented Software* (GAMMA et al, 1995), proliferaram-se diversos trabalhos propondo os mais variados tipos de diretrizes e padrões para o projeto e implementação de sistemas de software orientados a objetos. Apesar da grande difusão destes trabalhos, poucos na literatura focaram na proposição e definição de diretrizes e padrões para o projeto e implementação de elementos de tratamento de exceções.

A grande dificuldade em se definir padrões e diretrizes para código de tratamento de exceções é que a decisão de como dar suporte a cada um estes elementos varia de linguagem para linguagem (GARCIA et al., 2001). Desta forma, é muito difícil definir padrões e diretrizes de projeto e implementação de tratamento de exceção que sejam ao mesmo tempo independentes de linguagem de programação e suficientemente concretos para serem facilmente absorvidos por desenvolvedores em geral.

Wirfs-Brock (2006) propõe um conjunto de diretrizes básicas para o projeto de tratamento de exceções em sistemas de software orientado a objetos. As diretrizes definidas pela autora são independentes de linguagem, porém bastante abstratas, em um nível de abstração muito distante do nível de implementação. Diretrizes como “Trate suas exceções o mais próximo do problema quanto possível” e “Use exceções apenas para situações emergenciais” são muito vagas e pouco práticas para desenvolvedores, especialmente aqueles menos experientes.

Há ainda o fórum de discussões *Portland Pattern Repository* (PORTLAND PATTERN REPOSITORY, 2012), o qual apresenta um conjunto de descrições de padrões de estruturação de elementos de mecanismos de tratamento de exceção em linguagens orientadas a objetos. Os padrões descritos neste repositório são trabalhos em progresso que podem ser editados e discutidos livremente com comentários, exemplos e contra-exemplos dos usuários. Os padrões descritos no fórum *Portland Pattern Repository* também são bastante abstratos e distantes do nível de implementação.

Já McCune (2006) apresenta um conjunto de más práticas de implementação recorrentes em sistemas desenvolvidos em Java. O autor, no entanto, restringe sua discussão a um conjunto de práticas de implementação de tratamento de exceções que devem ser evitadas por desenvolvedores Java. Não se discutem quaisquer práticas que devem ser seguidas por desenvolvedores Java em geral a fim de construir sistemas de software mais tolerantes a falhas.

Chen et al. (2009) apresentam um conjunto de anomalias[3] relacionadas à estrutura do código de tratamento de exceções em Java, além de um conjunto de refatorações associadas às anomalias discutidas. Este é um dos poucos trabalhos em que, de fato, se apresentam e se discutem ações concretas de como reestruturar e melhorar o código de tratamento de exceções mal estruturado em aplicações existentes. Este trabalho, no entanto, sofre da mesma limitação dos trabalhos apresentados na Seção 2.1.1.1: assume-se que a aplicação possui um código de tratamento de exceções em que apenas ajustes pontuais são necessários. Além disso, não é apresentado a desenvolvedores em geral como é possível implementar código para tratamento de exceções que seja bem estruturado desde o princípio.

2.2. Sistemas de recomendação para a engenharia de software

Ainda que conceitos relacionados já viessem sendo discutidos em áreas correlatas, como ciências cognitivas (RICH, 1979) e *information retrieval* (SALTON, 1989), é dito que sistemas de recomendação consolidaram-se como área de pesquisa independente no começo dos anos 1990 com a publicação do

[3] Do inglês *Code Smells*

trabalho seminal sobre filtros colaborativos (GOLDBERG et al., 1992). De acordo com (ACM RECOMMENDER SYSTEM):

> *Sistemas de recomendação* são aplicações de software que almejam auxiliar usuários em suas tomadas de decisão enquanto interagem com um grande volume de informações.

O objetivo de um sistema de recomendação é prover itens de informação relevantes a um usuário a fim de tentar satisfazer sua necessidade em uma determinada tarefa e em um determinado contexto. No contexto de sistemas de recomendação, a necessidade de um usuário refere-se a um determinado tópico sobre o qual o usuário deseja adquirir conhecimento. A relevância de um item de informação, por sua vez, refere-se à medida do quanto um item atende à necessidade de um usuário.

Sistemas de recomendação podem ser divididos em sistemas colaborativos, sistemas baseados no conteúdo, sistemas baseados em conhecimento e sistemas híbridos. Sistemas de recomendação colaborativos exploram informações a respeito do comportamento e opiniões de uma comunidade de usuários a fim de prever quais itens um usuário corrente terá maior probabilidade de se interessar. Sistemas de recomendação baseados no conteúdo exploram as informações que descrevem determinados itens a fim de recomendar a um usuário outros itens com informações similares. Sistemas de recomendação baseados em conhecimento realizam suas recomendações a partir de um conjunto de requisitos definidos por um usuário, buscando por itens que atendam a estes requisitos.

O interesse por sistemas de recomendação rapidamente cresceu nos anos 1990, muito impulsionado pela Internet. A Internet mostrou-se um campo abundante para a exploração de novos serviços, como comércio eletrônico e redes sociais, além de uma grande fonte de informações e conhecimento usados para aprimorar ainda mais os sistemas de recomendação. Desde então, o uso de sistemas de recomendação vêm se tornando cada vez mais usual. Eles vêm ajudando pessoas a encontrar informações e tomar decisões em tarefas diversas, especialmente naquelas em que falta proficiência às pessoas envolvidas, ou naquelas em que não é possível se ter em mão todas as informações disponíveis.

A comunidade de Engenharia de Software também vem explorando o uso de sistemas de recomendação como forma de auxiliar engenheiros de software a melhor desempenhar suas atividades já há algum tempo. Trabalhos pioneiros (DEVANBU, BRACHMAN *et al.*, 1991; FISCHER, HENNINGER *et al.*, 1991;

MAAREK, BERRY *et al.*, 1991; RITTRI, 1989;) já exploravam conceitos de *information retrieval* e aprendizado de máquina para recomendar módulos reutilizáveis a engenheiros de software antes mesmo da consolidação do termo "sistema de recomendação".

Nos últimos anos, a crescente complexidade dos sistemas de software fez com que a quantidade de informação envolvida nas atividades da engenharia de software se tornasse cada vez maior. Mesmo tarefas aparentemente simples podem envolver a navegação, descoberta, compreensão e modificação dos artefatos desejados dentre um conjunto de diversos outros artefatos. Para se ter uma idéia, uma simples tarefa como alterar uma mensagem em um menu de contexto em um projeto de software como o Eclipse requer a descoberta e a modificação de um arquivo de código fonte Java, um arquivo de configuração XML e um arquivo de propriedades dentre um conjunto de mais de 10.000 artefatos distintos. Sem o suporte adequado, engenheiros de software podem perder-se em meio a artefatos irrelevantes e gastar um tempo desproporcional para realizar suas tarefas.

Neste contexto, Sistemas de Recomendação para a Engenharia de Software (SRESs) vêm sendo propostos e implementados em diversas outras atividades da engenharia de software além do reuso de software – de recomendações de engenheiros especialistas em determinado assunto a recomendações de relatórios de falhas previamente publicados em projetos *open source*. De acordo com Robillard, Walker e Zimmermann (2010, p. 81):

> Um ***Sistema de Recomendação para a Engenharia de Software*** é um sistema de software que provê informações que são consideradas valiosas em uma determinada tarefa da engenharia de software e em um dado contexto.

Dada a diversidade de atividades englobadas pela engenharia de software, também são diversas as possibilidades de propostas e implementações de SRESs. Por este motivo, é difícil generalizar uma arquitetura padrão para os SRESs. No entanto, é esperado que os SRESs provejam, no mínimo, três funcionalidades (ROBILLARD, WALKER e ZIMMERMANN, 2010):

- *Mecanismo de coleta de dados* para coletar e armazenar em bases de dados os dados relativos a uma determinada atividade da Engenharia de Software, além de outros artefatos que também sejam relevantes;

- *Mecanismo de recomendação* para analisar a base de dados e gerar as recomendações; e
- *Interface com o usuário* para acionar o mecanismo de recomendação e apresentar os resultados.

2.2.1. Trabalhos relacionados

2.2.1.1. Auxílio no uso de APIs e *frameworks*

O uso de *frameworks* e APIs (do inglês *Application Programming Interface*, ou interface para programação de aplicações) permite às equipes de desenvolvimento construir aplicações completas com menos esforço de implementação através do reuso de funcionalidades. Para tanto, é necessário que os desenvolvedores saibam como usar estes *frameworks* e APIs, i.e., é necessário saber quais classes devem ser estendidas, quais configurações devem ser feitas, quais métodos devem ser chamados, dentre outros. Neste contexto, alguns trabalhos propõem sistemas de recomendação para auxiliar desenvolvedores a se beneficiar mais facilmente das facilidades prometidas por *framework*s e APIs.

O sistema *CodeWeb* (MICHAIL, 2000) auxilia desenvolvedores a descobrir padrões de uso de *framework*s e APIs. Tais padrões são descritos em termos de classes e métodos que são freqüentemente utilizados conjuntamente e são coletados a partir de repositórios de aplicações baseadas em *frameworks* e APIs através de técnicas de mineração de dados. De posse desses padrões de uso, desenvolvedores têm um ponto de partida para explorar *frameworks* e APIs a fim de descobrir como concretizar uma determinada funcionalidade. Neste estudo, não foi realizado nenhum tipo de avaliação mais formal para o sistema *CodeWeb*. Apenas foi exemplificado um cenário de uso do sistema, mostrando como um desenvolvedor em tese poderia se beneficiar dos padrões de uso gerados pelo sistema

Mandelin *et al.* (2005) propõem uma técnica capaz de auxiliar desenvolvedores de aplicações baseadas em *framework*s e APIs em um cenário típico de desenvolvimento: o desenvolvedor sabe qual o tipo de objeto que necessita, mas não sabe como obter uma instância desse tipo. As necessidades do

desenvolvedor são modeladas através de relações do tipo T1 → T2, em que T1 indica um tipo de classe que se tem posse e T2 indica um tipo de classe que se deseja obter uma instância. A técnica proposta recomenda aos desenvolvedores como é possível obter uma instância do tipo T2 a partir de uma instância do tipo T1 através de possíveis seqüência de chamadas encadeadas de funções. Para tanto, repositórios de aplicações previamente implementadas são minerados a fim de se coletar as seqüência de chamadas que são recomendadas aos desenvolvedores. A técnica foi analisada através de dois experimentos. O primeiro foi um experimento analítico que coletou um conjunto de situações problemas do tipo "possuo um objeto do tipo T1 e desejo uma instância do tipo T2" e em seguida aplicou a técnica sobre cada problema coletado, registrando em qual posição a solução desejada foi encontrada. A solução desejada foi definida como "a solução mais concisa e eficiente possível se julgada por um ser humano" (Mandelin *et al.*, 2005, p 11). O segundo experimento designou para um grupo de desenvolvedores um conjunto de quatro problemas, dos quais dois deveriam ser resolvidos com o auxílio da técnicas e os outros dois sem o auxílio. Para cada usuário, o tempo e a precisão das repostas eram registrados e comparados ao fim do experimento.

Holmes, Walker e Murphy (2006) propõem auxiliar desenvolvedores de aplicações a aprender como usar *framework*s e APIs através do provimento de exemplos de código que demonstram na prática o uso destes *frameworks* e APIs. Os autores concretizam sua proposta com a implementação do sistema de recomendação *Strathcona*. A partir do código fonte em que o desenvolvedor está trabalhando, o sistema *Strathcona* extrai um conjunto de informações, como a assinatura do método e os tipos e identificadores das variáveis usadas, e realiza uma busca heurística em um repositório de exemplos previamente populado. O sistema foi avaliado através de uma série de quatro experimentos. O primeiro experimento designou para um grupo de desenvolvedores um conjunto de três tarefas de programação típicas, as quais deveriam ser resolvidas com o auxílio da ferramenta *Strathcona*. O uso da ferramenta era monitorado durante o experimento para avaliar se e de que maneira os usuários usavam as recomendações realizadas. O segundo experimento comparou o resultado das recomendações realizadas pela ferramenta *Strathcona* com ferramentas típicas de busca textual, como `grep`. O terceiro experimento baseou-se em análises de precisão da ferramenta. A partir de um método selecionado aleatoriamente, a

ferramenta extraía informações deste método e as usava para realizar a recomendação, verificando-se em qual posição o método original era retornado dentre as recomendações realizada pela ferramenta. O último experimento monitorou o uso da ferramenta por um desenvolvedor em seu ambiente profissional de trabalho, analisando de que maneira as recomendações foram utilizadas pelo desenvolvedor.

Bruch, Mezini e Monperrus (2010) propõem uma abordagem para extrair documentação auxiliar de *framework*s a partir do código de aplicações clientes. Na abordagem proposta, são extraídos quatro tipos distintos de diretivas de uso de *frameworks*, as chamadas *sub classing directives*. Tais diretrizes são definidas em termos de: (i) quais métodos devem ser sobrescritos pela aplicação cliente, (ii) quais métodos que estão sobrescrevendo um método do *framework* também devem invocar a implementação do método sobrescrito, (iii) quais métodos devem ser invocados no contexto de métodos sobrescritos pela aplicação e (iv) quais métodos costumam ser sobrescritos conjuntamente. Estas diretivas podem então ser usadas por desenvolvedores de aplicações como fonte de informação auxiliar e complementar a documentação base provida por *framework*s e APIs. Também podem ser usadas por desenvolvedores de *frameworks* a fim de analisar de que forma os pontos de extensão do *frameworks* estão sendo de fato estendidas pelas aplicações clientes. A abordagem foi avaliada através de um experimento em que se comparavam as documentações geradas pela abordagem a partir de um conjunto de aplicações e a documentação provida originalmente pelos *frameworks* e APIs.

2.2.1.2. Auxílio ao reuso de código

O desejo de reutilizar artefatos de software é quase tão antigo quanto a própria Engenharia de Software, remontando a meados de 1970 quando Parnas (1976) propôs o conceito de famílias de software. A fim de alcançar o desejado reuso de artefatos de software, é primordial a disposição de mecanismos que possibilitem a busca por artefatos que possam ser reutilizados. De acordo com a revisão bibliográfica realizada por Mili *et al.* (1998), as primeiras propostas de mecanismos e ferramentas para recomendação de código reutilizável

apresentavam uma limitação em comum. Nestas propostas os desenvolvedores decidiam usar as ferramentas disponíveis apenas quando sabiam antecipadamente da existência de módulos que poderiam ser reutilizados. Neste contexto, alguns sistemas de recomendação vêm sendo propostos com o objetivo de auxiliar desenvolvedores a encontrar fragmentos de código que sejam reutilizáveis, focando em uma melhor interação entre desenvolvedores e sistemas de buscas.

Hill e Rideout (2004) extrapolam o conceito de auto-preenchimento[4] presente em ambientes de desenvolvimento típicos para, ao invés de complementar apenas nomes de métodos e variáveis, o auto-preenchimento ser capaz de complementar o corpo de um método através de técnicas de aprendizado de máquina. Ao iniciar a escrita do corpo de um método a técnica *Automatic Method Completion* proposta pelos autores tenta recomendar um possível preenchimento para o restante do corpo do método. Para tanto, a técnica representa o método em desenvolvimento e os seus identificadores como um vetor multidimensional. Este vetores são comparados com outros vetores previamente computados e aquele que apresentar a melhor correspondência é recomendado. A técnica foi avaliada através de experimentos em que os autores proviam métodos incompletos, além de três possíveis preenchimentos para estes métodos, classificados em "correto", "bom" e "medíocre". Em seguida, aplicava-se a técnica ao método incompleto e comparava-se a recomendação realizada aos complementos criados pelos autores. Os possíveis complementos providos pelos autores mais a recomendação realizada eram então ranqueados de acordo com critérios subjetivos dos próprios autores, i.e., o quão "bom" eles achavam cada um dos complementos.

Ye e Fischer (2005) propuseram a integração do sistema de recomendação ao ambiente de desenvolvimento de software a fim de tornar os desenvolvedores mais cientes das oportunidades de reuso através de um mecanismo de recomendação proativo. Para avaliar o conceito proposto, os autores implementaram a ferramenta *CodeBroker*. Tal ferramenta mantém um processo em segundo plano que monitora a atividade de uso do ambiente de desenvolvimento. Após detectar a escrita de novos comentários ou assinaturas de método, recomendações de reuso são realizadas através de uma busca em um

[4] Do inglês *auto-completion*

repositório de código reutilizável. As buscas são feitas com base nas informações extraídas dos comentários e assinatura de funções. A ferramenta *CodeBroker* foi analisada através de experimentos analíticos que mediram métricas de precisão e revocação, além de experimentos com desenvolvedores simulando cenários de uso da ferramenta.

2.3. Estudos de qualidade de código de tratamento de exceções

No contexto desta dissertação, foram realizados alguns estudos exploratórios com o objetivo de compreender melhor como o tratamento de exceções é realizado em aplicações reais usadas na indústria. Para tanto, inspeções manuais do código fonte de aplicações foram realizadas.

No primeiro estudo (BARBOSA e GARCIA, 2010), foram inspecionadas versões de um sistema de apoio a logística de insumos da indústria petrolífera. Ao todo, foram inspecionadas 7 versões deste sistema, compreendendo um período de evolução de aproximadamente 2 anos. A inspeção destas versões revelou alguns fenômenos interessantes.

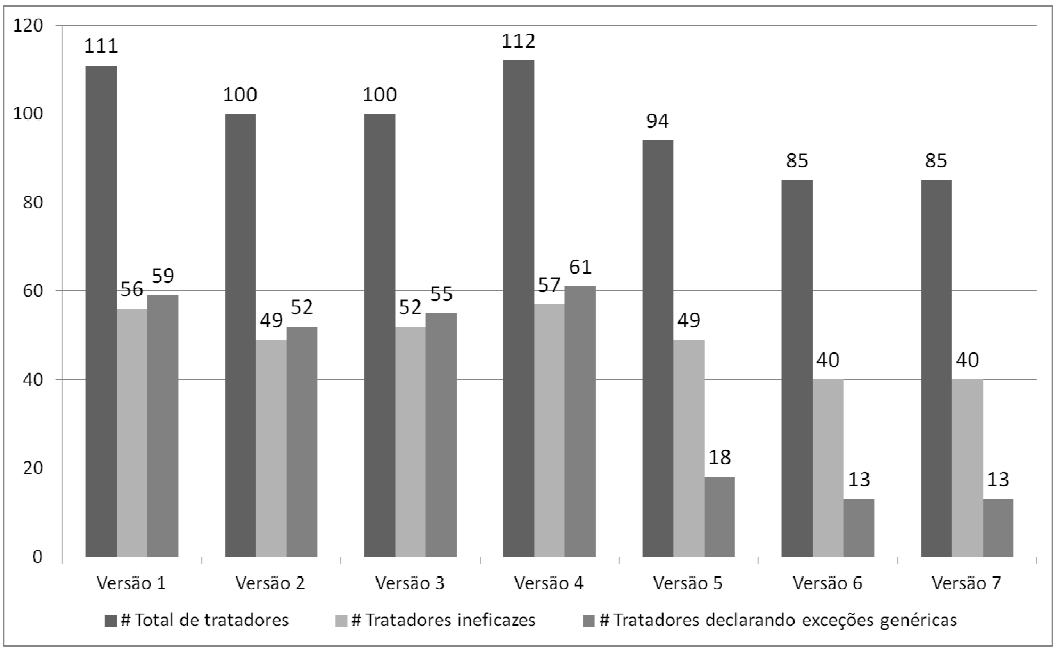

Figura 4 Evolução de tratadores

Como pode ser observado no gráfico da Figura 4, nas 4 primeiras versões do sistema cerca de 50% dos tratadores implementados eram tratadores ineficazes, como tratadores vazios, tratadores que apenas imprimem a pilha de execução

associada à exceção, dentre outros. Ainda em relação às 4 primeiras versões do sistema, também cerca de 50% dos tratadores implementados declaravam tipos de exceções muito genéricos, como *Throwable* e *Exception*. A implementação de tratadores que declaram tipos de exceções muito genéricos aumenta o risco de capturar inadvertidamente exceções indesejadas por subsunção. A captura de exceções indesejadas por subsunção está relacionada com algumas falhas recorrentes em sistemas de software (Coelho *et al.*, 2008) e, por este motivo, a implementação de tratadores declarando tipos de exceções muito genéricos deve ser realizado com cuidado. Percebe-se assim que o uso de tratadores ineficazes e tratadores declarados com tipos muito genéricos foi bastante comum nas 4 primeiras versões do sistema analisado. Este cenário sugere um certo descuido com a implementação do tratamento de exceções nas primeiras versões do sistema.

Nas últimas 3 versões do sistema, quando este já havia atingido certa maturidade em suas funcionalidades principais, observou-se uma tentativa em melhorar a qualidade do código do tratamento de exceções. Mas, como pode ser visto na Figura 4, observaram-se apenas esforços em termos de tornar as exceções declaradas nos tratadores mais específicas. Se nas 4 primeiras versões analisadas a porcentagem de tratadores declarando tipos de exceção muito genéricos era de cerca de 50%, nas últimas 3 versões este valor caiu para cerca de 20%. No entanto, o mesmo comportamento não foi observado em relação aos tratadores ineficazes. A porcentagem de tratadores ineficazes permaneceu em torno de 50% nas últimas 3 versões analisadas. Percebe-se assim que para o sistema analisado não houve um maior esforço por parte dos desenvolvedores em termos de implementar ações de recuperação que tentassem recuperar o estado interno do sistema para um estado válido.

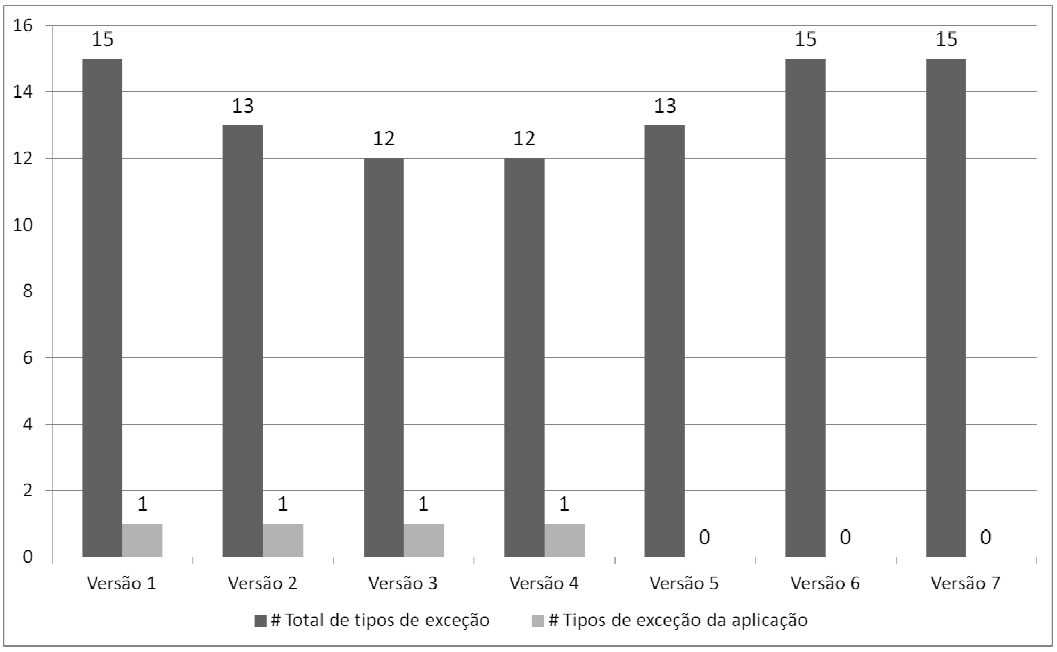

Figura 5 Evolução de tipos de exceção

Como mostra a Figura 5, observou-se ainda que dentre os tipos de exceções usados nas 4 primeiras versões do sistema, mais de 90% eram tipos de exceções definidos por APIs. Já nas 3 versões finais, todos os tipos de exceções usados pelo sistema eram definidos por APIs.

No segundo estudo realizado (BARBOSA e GARCIA, 2011), foram analisados dois *frameworks* open-source desenvolvidos e mantidos pelo Google: Guice[5] e GWT[6]. Ao todo foram inspecionadas 10 versões do primeiro *framework* e 7 versões do segundo, compreendendo períodos de 3 e 2 anos, respectivamente. Assim como no primeiro estudo, neste segundo estudo também se observou que o código de tratamento de exceções nas primeiras versões dos sistemas analisados era de má qualidade. E tal qual o estudo anterior, apenas melhorias superficiais foram realizadas ao longo da evolução do sistema, sendo poucos esforços demandados para a melhoria das ações de recuperação implementadas pelos tratadores. Neste segundo estudo, em especial, observou-se que um dos sistemas analisados apresentava projeto de tratamento de exceções tão inadequado nas versões iniciais do sistema que sua própria arquitetura de software sofreu instabilidades durante sua evolução. Alguns módulos implementando tratamento

[5] http://code.google.com/p/google-guice/

[6] http://developers.google.com/web-toolkit/

de exceções no primeiro *framework* foram completamente removidos do projeto do sistema e reescritos do zero.

A conclusão dos estudos exploratório preliminares realizados mostra que mesmo desenvolvedores experientes têm dificuldade, ou relutam, em implementar código de tratamento de exceções. Em especial, percebe-se grande dificuldade em definir e implementar ações de recuperação que sejam eficazes. Além disso, a prática de deixar para versões futuras do sistema a correta implementação do tratamento de exceções não é adequada. Em muitos casos, deixar para depois pode ser tarde demais. Faz-se então necessário que desenvolvedores em geral adotem outra prática de desenvolvimento no que concerne o tratamento de exceções. É necessário que eles realizem a correta implementação do tratamento de exceções desde as primeiras versões de um sistema.