Modularização de Tratamento de Exceções Usando Programação Orientada a Aspectos

Raquel de Albuquerque Maranhão Ferreira

Trabalho Final de Mestrado Profissional em Computação

Instituto de Computação
Universidade Estadual de Campinas

Modularização de Tratamento de Exceções Usando Programação Orientada a Aspectos

Raquel de Albuquerque Maranhão Ferreira

24/02/2006

Banca Examinadora:

- **Profª. Drª. Cecília Mary Fischer Rubira (Orientadora)**
  Instituto de Computação - UNICAMP
- **Profª. Drª. Ana Cristina Vieira de Melo**
  Departamento de Ciência da Computação, IME - USP
- **Profª. Drª. Ariadne Maria B. Rizzoni Carvalho**
  Instituto de Computação - UNICAMP
- **Profª. Drª. Eliane Martins (Suplente)**
  Instituto de Computação – UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Ferreira, Raquel de Albuquerque Maranhão

F413m Modularização de tratamento de exceções usando programação orientada a aspectos / Raquel de Albuquerque Maranhão Ferreira -- Campinas, [S.P. :s.n.], 2006.

Orientadora : Cecília Mary Fischer Rubira

Trabalho final (mestrado profissional) - Universidade Estadual de Campinas, Instituto de Computação.

1. Programação orientada a aspectos. 2. Tolerância a falha (Computação). 3. Engenharia de Software. I. Rubira, Cecília Mary Fischer. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Modularização de Tratamento de Exceções Usando Programação Orientada a Aspectos

Este exemplar corresponde à redação final do Trabalho Final devidamente corrigido e defendido por Raquel de Albuquerque Maranhão Ferreira e aprovado pela Banca Examinadora.

Campinas, 24 de Fevereiro de 2006.

Profª. Drª. Cecília Mary Fischer Rubira
(Orientadora)

Trabalho Final apresentado ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Computação na área de Engenharia de Computação.

TERMO DE APROVAÇÃO

Trabalho Final Escrito defendido e aprovado em 24 de fevereiro de 2006, pela Banca Examinadora composta pelos Professores Doutores:

Ariadne M. B. R. Carvalho

Profa. Dra. Ariadne Maria Brito Rizzoni Carvalho
IC - UNICAMP

Profa. Dra. Ana Cristina Vieira de Melo
IME - USP

Cecília Mary Fischer Rubira

Profa. Dra. Cecília Mary Fischer Rubira
IC- UNICAMP

Ao meu querido esposo, Ilco

Agradecimentos

Ao meu esposo, Ilco, pelo inabalável companheirismo, apoio e dedicação.

À minha orientadora, Profª. Drª. Cecília Mary Fischer Rubira, pela oportunidade e confiança depositada.

Ao amigo, Fernando Castor Filho, que foi peça fundamental para a realização deste trabalho. Agradeço-lhe pela extrema dedicação e valiosos ensinamentos.

A todos os colegas e amigos da Unicamp, em especial aos sempre presentes Kacuta e Joyce, pelo companheirismo e força dedicados.

Aos meus amigos, por compreenderem minhas tão constantes ausências para dedicar-me aos estudos.

Ao Instituto de Computação da Unicamp pela oportunidade oferecida para a realização de um desejo há muito acalentado.

E, finalmente à minha mãe, Maria José, que sempre batalhou para que eu alcançasse meus objetivos. Dedico-lhe esta obra como reconhecimento da sua participação na pessoa que hoje eu sou.

Resumo

Programação Orientada a Aspectos (POA) tem sido considerada uma abordagem interessante para modularizar o comportamento excepcional de um sistema. Porém, ainda existem algumas questões em aberto sobre o uso de POA com este objetivo. Nenhum trabalho na literatura tentou verificar se POA realmente promove melhorias em atributos de qualidade quando usada para modularizar código de tratamento de exceções não-trivial, objetivando: avaliar a escalabilidade dos aspectos ao modularizar tratamento de exceções em sistemas com um número significativo de tratadores; e avaliar a interação entre aspectos de tratamento de exceções e aspectos de outros interesses como, por exemplo, distribuição, persistência, segurança etc.

Este trabalho apresenta um estudo quantitativo da adequação de POA para modularizar código de tratamento de exceções em sistemas que possuem um número significativo de tratadores não-triviais, e também em sistemas nos quais interesses transversais diferentes de tratamento de exceções, no caso distribuição e persistência, já foram modularizados com aspectos. Este estudo consistiu na refatoração de dois sistemas orientados a objetos e um sistema orientado a aspectos, que tiveram seu código responsável pelo tratamento de exceções movido para aspectos. Foi utilizado um conjunto de métricas para avaliar atributos de qualidade das versões original e refatorada desses sistemas. Observou-se que POA promoveu a separação de interesses entre o código de tratamento de exceções e o código normal dos sistemas. Porém, contradizendo a intuição geral, a versão orientada a aspectos desses sistemas não apresentou ganhos significativos para as métricas de tamanho empregadas.

Abstract

Aspect-Oriented Programming (AOP) has usually been considered as an approach to modularize the exceptional behavior of a system. However, there are questions related to possible trade-offs involved in using AOP with this objective that are not yet well known. To the best of our knowledge, no work in literature has attempted to assess whether AOP really promotes an enhancement in well-understood quality attributes, when used for modularizing nontrivial exception handling code with focus on: evaluation of the scalability of aspects for modularizing exception handling in systems with a significant number of handlers; and evaluation of interactions between exception handling aspects and aspects implementing other concerns like distribution and persistence.

This work presents a quantitative study of the adequacy of AOP for modularizing exception handling code in systems with a significant number of nontrivial handlers, and also in systems possessing aspects implementing other concerns. The study consisted of refactoring two object-oriented and one aspect-oriented systems so that the code responsible for handling exceptions was moved to aspects. It was employed a suite of metrics to measure quality attributes of the original and refactored systems. It was found that AOP improved separation of concerns between exception handling code and normal application code. However, contradicting the general intuition, the aspect-oriented version of the system did not present significant gains for any of size metrics employed.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

INTRODUÇÃO

Na sociedade moderna, os sistemas de software são imprescindíveis em quase todas as atividades, inclusive nas consideradas mais críticas, como por exemplo automação bancária, sistemas de controle automobilísticos, sistemas comerciais etc. Esses são exemplos atuais de áreas onde essa "dependência" (Do inglês: *dependability*) de sistemas de software já ocorre e está tornando-se cada vez mais crítica. Esses sistemas fortemente ligados a atividades essenciais são denominados **sistemas críticos**, pois a sua interrupção ou funcionamento inadequado representam riscos para vidas humanas, bens patrimoniais ou meio-ambiente. Um dos principais requisitos desses sistemas é a confiabilidade [22].

Tratamento de exceções [2] é um mecanismo bastante conhecido, utilizado para introduzir recuperação de erros por avanço (Do inglês: *forward error recovery*) em sistemas de software. O uso de tratamento de exceções para estruturar a atividade excepcional de um sistema é uma prática popular para construir sistemas nos quais confiabilidade é um requisito essencial. A maioria das linguagens de programação modernas, tais como C++ [44], Java [45], C# [46], Ada [47], Eiffel [48] e Smalltalk [49], incluem mecanismos de tratamento de exceções como uma de suas características. No desenvolvimento de software, grande parte do código de um sistema é destinado à detecção e tratamento de exceções [2]; porém, essa parte do código normalmente é pouco entendida, testada e documentada. Desta forma, em muitos casos, a maioria das falhas de projeto [22] que ocorrem em um sistema estão localizadas justamente no código responsável pela sua atividade excepcional [2], cujo objetivo é proporcionar robustez ao sistema.

Programação Orientada a Aspectos (POA) [3] é um paradigma recente que está se consolidando como um próximo passo na evolução das linguagens de programação orientadas a objetos. POA apresenta-se como um mecanismo para modularizar sistemas que apresentam **interesses transversais** [43] (Do inglês: *crosscutting concerns*), ou seja, interesses cuja implementação está espalhada em diversas partes do sistema, misturada a código responsável por outros requisitos. Interesses transversais não podem ser totalmente

modularizados por técnicas de estruturação unicamente orientadas a objetos, como padrões de projeto [27]. Existem vários trabalhos na literatura que mostram que POA é útil para modularizar diversos requisitos nos sistemas, como: distribuição [36, 37], persistência [4, 36], autenticação [4], tratamento de exceções [26, 37] e implementação de padrões de projeto [34]. Devido ao forte impacto que POA tem provocado na indústria e também no meio acadêmico, em Janeiro/2001 a revista Technology Review incluiu POA na lista das "dez tecnologias emergentes que transformarão o mundo" [40].

Diversas técnicas populares para construção de sistemas de software confiáveis, como tratamento de exceções e transações atômicas [41] são inerentemente transversais [4, 26, 36, 37]. Sendo assim, acredita-se que POA possa ser capaz de ajudar na construção de sistemas de software confiáveis.

1.1 Problema

Alguns estudos já foram realizados com o intuito de avaliar o quão lucrativo é utilizar POA para modularizar tratamento de exceções. Dentre esses estudos, pode-se ressaltar o estudo de Lippert & Lopes [26] e o de Castor & Rubira & Garcia [16].

O primeiro [26] apresentou um estudo de caso no qual o *framework* orientado a objetos JWAM foi refatorado para verificar a eficácia de utilizar aspectos para separar o código responsável pelo tratamento de exceções do código normal do sistema. As limitações principais desse estudo foram: (i) o sistema-alvo do estudo apresentava código de tratamento de exceções genérico; (ii) a análise qualitativa baseou-se em atributos de qualidade não muito estabelecidos; e (iii) a análise quantitativa baseou-se apenas em medidas de linhas de código (Do inglês: *Lines of Code* - LOC).

O segundo estudo [16] também apresentou um trabalho experimental que refatorou parte de um sistema orientado a objetos real utilizando aspectos para separar o código de tratamento de exceções do código normal do sistema. Além disso, nesse estudo houve a iniciativa de verificar mais precisamente a qualidade do software gerado, através do uso de um conjunto de métricas [1] de qualidade, adequado para sistemas orientados a aspectos. As principais limitações desse estudo são: (i) o estudo teve foco em apenas uma linguagem de POA,

AspectJ [4]; (ii) as possíveis estratégias para implementação de comportamento excepcional de sistemas foram parcialmente cobertas; (iii) não foi avaliada a escalabilidade de AspectJ para um número significativo de tratadores; e (iv) não foi avaliada a integração entre aspectos de tratamento de exceções e aspectos de outros interesses.

Modularizar tratamento de exceções com POA é uma abordagem recente e ainda não consolidada. Para melhor ilustrar esse fato, veja o que o "AspectJ guru" Ron Bodkin, respondendo um e-mail que pedia orientações sobre como usar POA para estruturar melhor o código de tratamento de exceções, disse:

"*I often use AOP to handle exceptions and find it valuable. At the same time, there are definitely some cases where exceptions and their handling are *not* crosscutting, and it is helpful to explicitly throw them and to explicitly handle them in a method. There are also cases where you might start with inline exception handling and then refactor into an aspect as you realize there is a common policy that can be captured.*"

Percebe-se que as opiniões sobre esse tema ainda giram em torno de tópicos abstratos, faltando definições mais concretas sobre técnicas e ferramentas que melhor auxiliem essa abordagem.

Apesar dos estudos sobre esse tema, ainda existem pontos em aberto sobre benefícios e danos que podem ser inseridos no sistema com a utilização de POA na refatoração de tratamento de exceções. Por exemplo, seria importante verificar o que ocorre em estudos similares quando o número de tratadores de exceções é bastante significativo e verificar o quão difícil é integrar aspectos dedicados a tratamento de exceções com aspectos relacionados a outros interesses, como por exemplo, distribuição, persistência, segurança etc.

1.2 Solução Proposta

O trabalho proposto baseia-se em pesquisas recentes nas seguintes áreas: Tratamento de Exceções, Separação de Interesses (Do inglês: *Separation of Concerns*) e Programação Orientada a Aspectos.

Este trabalho propõe um estudo sobre modularização do tratamento de exceções com POA, com foco na verificação da escalabilidade dessa abordagem, à medida que o número de tratadores de exceções aumenta, e na integração dos aspectos para tratamento de exceções com aspectos relacionados a outros interesses, como por exemplo, distribuição, persistência, segurança etc. Este trabalho inclui três estudos de caso, que foram executados de acordo com os seguintes passos:

- ✓ Seleção de um sistema que se enquadre em uma das seguintes opções: (i) implementado em Java e contendo quantidade expressiva de tratadores de exceções (verificação da escalabilidade); ou (ii) implementado em AspectJ e possuindo aspectos de interesses distintos de tratamento de exceções (verificação da integração entre aspectos).
- ✓ Refatoração do código original, que consiste em mover para aspectos todas as ocorrências de código responsável pelo tratamento de exceções que se encontram misturadas ao código responsável pela atividade normal do sistema.
- ✓ Coleta de métricas para quantificar o impacto na qualidade do sistema ao utilizar aspectos para separar o código de tratamento de exceções do código normal do sistema, através de um conjunto de métricas (cujos detalhes podem ser vistos em [1]) capaz de avaliar implementações orientadas a aspectos.
- ✓ Finalmente, as medidas coletadas são analisadas para que conclusões baseadas nesses resultados possam ser obtidas.

1.3 Trabalhos Relacionados

Kienzle & Guerraoui [42] realizaram um dos primeiros estudos sobre a aplicabilidade de POA para desenvolver sistemas confiáveis. Os autores empregaram transações atômicas como representantes dos mecanismos de tolerância a falhas, e AspectJ como representante de linguagens orientadas a objetos. Eles concluíram que POA não é útil para modularizar tolerância a falhas nessas condições.

O estudo Soares & Laureano & Borba [36] conclui que, embora a linguagem AspectJ apresente algumas limitações, ela ajuda a modularizar a execução transacional de métodos em várias situações que ocorrem em sistemas reais.

Lippert & Lopes [26] realizaram o mais conhecido estudo sobre o uso de POA para tratamento de exceções. Esse estudo teve como principal objetivo avaliar se POA poderia ser utilizada para separar o código responsável por detectar e tratar exceções do código de atividade normal de um sistema. Para tal, foi realizado um experimento prático em um grande *framework* orientado a objetos, no qual todo o código destinado a tratamento de exceções foi movido para aspectos. Ao final, os autores desse estudo concluíram que POA trouxe vários benefícios, como maior separação entre os interesses no decorrer das linhas de código e redução drástica no número de linhas de código (LOC).

Castor & Rubira & Garcia [16] realizaram um estudo mais recente cujo foco também foi o uso de POA para tratamento de exceções. Nesse estudo, os autores tiveram a preocupação de realizar uma análise quantitativa mais rígida sobre a qualidade do sistema antes e após a "aspectização", baseada em um conjunto de métricas apropriadas para POA. Esse estudo obteve conclusões que reforçam o estudo de Lippert & Lopes [26], como a obtenção de maior separação de interesses no código do programa, mas também apontou para conclusões divergentes, como o questionamento sobre a intuição de que aspectos ajudam a produzir programas menores. O estudo apresentado em [16] serviu como motivação para o tema do presente trabalho, que intenta continuar as pesquisas ali iniciadas e desenvolver os seguintes pontos: (i) verificar a escalabilidade de AspectJ com relação à "aspectização" de tratadores de exceções, ou seja, o que ocorre em estudos similares quando o número de tratadores aumenta bastante (aplicado no primeiro e segundo estudos de caso deste trabalho); e (ii) verificar o quão difícil é integrar aspectos dedicados a tratamento de exceções com aspectos de demais interesses, por exemplo, distribuição e persistência (aplicado no terceiro estudo de caso deste trabalho).

1.4 Estrutura do Trabalho

O restante deste documento está organizado da seguinte forma:

- ✓ **Capítulo 2 – Fundamentos de Engenharia de Software:** Para embasar este trabalho, o segundo capítulo apresenta fundamentos de tratamento de exceções e programação orientada a aspectos. Além disso, esse capítulo apresenta o conjunto de métricas de qualidade usado para avaliar os experimentos práticos deste trabalho.
- ✓ **Capítulo 3 - Tratamento de Exceções com POA em Sistemas Orientados a Objetos:** Este capítulo descreve dois dos três experimentos práticos realizados neste trabalho. O primeiro consiste no estudo de caso realizado com o sistema orientado a objetos *Java Pet Store*, que é disponibilizado pela Sun Microsystems como exemplo de sistema para a plataforma Java 2 Enterprise Edition (J2EE) [9]. Já o segundo corresponde ao experimento prático realizado em um sistema de informações real, o Telestrada. Estes estudos de caso tiveram o enfoque de refatorar o sistema para "aspectizar" o código de tratamento de exceções e coletar métricas das versões original e refatorada, possibilitando a análise e obtenção de conclusões a partir desses resultados.
- ✓ **Capítulo 4 - Tratamento de Exceções com POA em Sistemas Orientados a Aspectos:** Neste capítulo, é descrito o terceiro experimento prático realizado neste trabalho. Ele consiste no estudo de caso realizado em um sistema de informações orientado a aspectos real, o *Health Watcher*. Este estudo teve o mesmo enfoque que os dois estudos anteriores, apresentando algumas peculiaridades geradas pelo fato de ser um sistema originalmente orientado a aspectos.
- ✓ **Capítulo 5 - Cenários de Tratamento de Exceções com POA:** Este capítulo ilustra alguns dos cenários que foram encontrados nesses três experimentos práticos e que merecem ser destacados.
- ✓ **Capítulo 6 – Conclusões e Trabalhos Futuros:** Finalmente, este capítulo sintetiza as contribuições deste trabalho e sugere trabalhos futuros.

Capítulo 2

FUNDAMENTOS DE ENGENHARIA DE SOFTWARE

Este capítulo define os conceitos e a terminologia utilizados neste trabalho. Na Seção 2.1 é apresentada uma visão geral sobre o conceito de tratamento de exceções, que é necessário para a compreensão deste trabalho. Neste trabalho são adotadas as terminologias de tolerância a falhas propostas por Lee e Anderson em [22]. Na Seção 2.2, explica-se em linhas gerais a programação orientada a aspectos, bem como são ilustrados conceitos básicos da linguagem orientada a aspectos AspectJ utilizada em experimentos práticos realizados neste trabalho. A Seção 2.3 apresenta um conjunto de métricas de qualidade para sistemas orientados a aspectos, utilizado para avaliar os experimentos práticos realizados.

2.1 Tratamento de Exceções

Sistemas computacionais são formados por componentes de *hardware* e *software* que eventualmente podem falhar. Tais **falhas** são provocadas por fenômenos naturais de origem interna ou externa e ações humanas acidentais ou intencionais. Quando uma falha ocorre em um sistema, sua manifestação dá origem a um ou mais **erros**, ou seja, inconsistências no estado do sistema. Um erro pode causar a ocorrência de **defeitos**, isto é, desvios do comportamento esperado para o sistema. A Figura 1, extraída de [22], fornece uma visão geral da relação entre falhas, erros e defeitos.

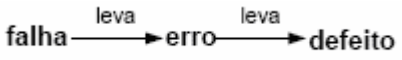

Figura 1 – Falha, erro e defeito [22]

Quando um programa (em um sentido geral: uma rotina, um componente de software, um sistema inteiro, etc.) recebe a solicitação de um serviço e produz uma resposta de acordo com a sua especificação, essa resposta é dita **normal**. Se o programa produz uma resposta

que não está em conformidade com sua especificação, esta resposta é considerada **anormal** ou **excepcional**. Respostas anormais normalmente indicam a ocorrência de um erro e, como se espera que essas respostas raramente ocorram, elas são chamadas de **exceções**. Quando exceções ocorrem, o programa deve ser capaz de tratá-las de forma a retornar a um estado coerente. A parte do comportamento de um programa que é responsável por tratar exceções é chamada de **atividade anormal ou excepcional**. Por outro lado, a parte do comportamento de um programa que é responsável pela funcionalidade especificada é chamada de **atividade normal**.

Tratamento de exceções [2] é uma técnica para estruturar a atividade excepcional de um sistema, de modo que erros possam ser detectados, sinalizados e tratados. Tratamento de exceções é uma técnica efetiva e largamente utilizada para incorporar tolerância a falhas em sistemas de software. Estima-se que entre 1 e 5% do código de sistemas de software modernos seja dedicado ao tratamento de exceções e essa proporção cresce com o tamanho e a idade desses sistemas [23].

Mecanismos de tratamento de exceções de linguagens de programação possuem suas características próprias, mas essencialmente eles seguem um mesmo processo. Após a identificação de um erro, uma exceção é levantada indicando a presença de uma inconsistência no estado do sistema. A partir deste momento, um **tratador** para a exceção é procurado pelo próprio mecanismo de tratamento de exceções da linguagem de programação. Caso um tratador para a exceção que foi levantada seja encontrado, ele é responsável por tentar levar o sistema para um estado livre de erros ou garantir que o sistema não falhe de maneira catastrófica.

2.1.1 Componente Tolerante a Falhas Ideal

O conceito de **componente tolerante a falhas ideal** (CTFI) [22] define um *framework* conceitual para estruturar tratamento de exceções em sistemas de software. Um CTFI é um componente no qual as partes responsáveis pela atividade normal e pela atividade anormal estão separadas e bem-definidas, dentro da sua estrutura interna, permitindo que erros sejam detectados e tratados com maior facilidade. O objetivo da abordagem de CTFI é

proporcionar uma forma de estruturar sistemas que minimize o impacto dos mecanismos de tolerância a falhas em sua complexidade global.

A Figura 2 apresenta a estrutura interna de um componente tolerante a falhas ideal e os tipos de mensagem que ele troca com outros componentes na arquitetura. Ao receber uma requisição de serviço, um CTFI fornece uma **resposta normal** se a requisição é processada com sucesso. Se a requisição de serviço não é válida, é levantada uma **exceção de interface**. Se o serviço está disponível mas ocorre uma falha durante o processamento da requisição e o CTFI não é capaz de tratá-la internamente, é levantada uma **exceção interna**, a ser tratada pela atividade anormal do componente. Se for possível levar o componente de volta a um estado consistente, sua atividade normal é retomada. Caso contrário, uma **exceção de defeito** é levantada.

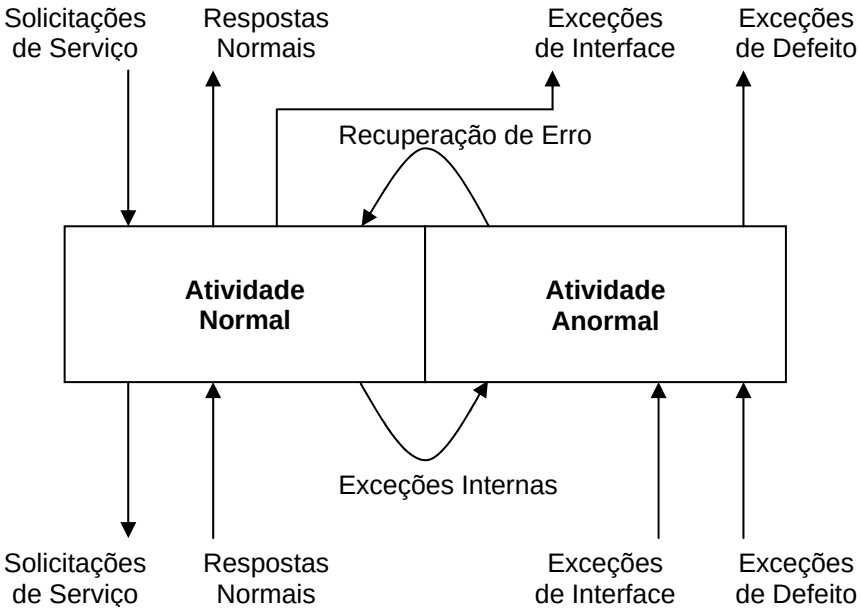

Figura 2 – Componente Tolerante a Falhas Ideal [22]

2.1.2 Tratamento de Exceções em Java

As exceções em Java [14] são representadas por classes que herdam da classe `java.lang.Throwable` (Figura 3). Quando uma exceção é lançada, uma instância da classe `Throwable` é criada. Duas subclasses diretas de `Throwable` são: `Error` e `Exception`. Em Java, programadores podem criar suas próprias classes de exceção, mas

é importante que elas sejam criadas de acordo com a convenção existente, isto é, devem ser subclasses de `Exception` (ao invés de subclasses diretas de `Throwable`). A classe `Error` é usada para indicar a ocorrência de problemas graves, ou seja, situações anormais, provocadas pelo ambiente operacional ou pela própria estrutura do programa, que o sistema não deve tentar tratar. Alguns exemplos de subclasses de `Error` são: `AssertionError`, `NoSuchMethodError`, `StackOverflowError`, e `OutOfMemoryError`.

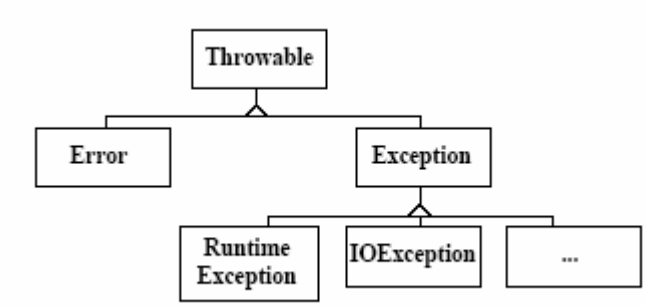

Figura 3 – Hierarquia de Exceções em Java

Do ponto de vista de verificações realizadas estaticamente pelo compilador de Java, pode-se identificar dois tipos de exceção na linguagem:

- ✓ **Exceções verificadas** são usadas para modelar falhas previstas no projeto de um sistema e, em geral, contornáveis. Devem sempre ser declaradas nas interfaces dos métodos que as lançam e precisam ser tratadas pelos métodos que as recebem, a menos que sejam lançadas também por estes últimos. São consideradas exceções verificadas todas aquelas que são subclasses de `java.lang.Exception` e que não são subclasses de `java.lang.RuntimeException`.
- ✓ **Exceções não-verificadas** são usadas para modelar falhas imprevistas, normalmente decorrentes de erros de programação. Normalmente estão relacionadas com inconsistências no código detectadas durante a execução do programa pela JVM (*Java Virtual Machine*). Por exemplo, a exceção de tempo de execução `NullPointerException` é levantada quando uma variável não-inicializada é referenciada. Exceções não-verificadas não precisam ser explicitamente declaradas nas interfaces dos métodos que as lançam e nem tratadas pelos métodos que as recebem, já que o compilador de Java não realiza nenhuma verificação estática para

essas exceções. São consideradas exceções não-verificadas todas aquelas que são subclasses de `java.lang.RuntimeException` ou de `java.lang.Error`. Exceções não-verificadas são usadas pela própria linguagem para sinalizar condições de erro e normalmente não são criadas. Elas recebem apenas tratamento genérico para evitar que o sistema falhe de maneira catastrófica.

Alguns conceitos importantes relacionados ao tratamento de exceções em Java são discutidos a seguir.

Lançamento de Exceções

Uma exceção é lançada usando-se a palavra-chave `throw` seguida do objeto correspondente à exceção ou uma referência a ele. Exemplo:

```
...
if (b = = 0) throw new DivByZero();
return a/b;
...
```

Quando um método pode lançar uma exceção verificada, deve-se usar a palavra-chave `throws` seguida do tipo de exceção na declaração do método. Se a exceção não for verificada, ainda é possível declará-la na interface do método, mas o compilador Java não verifica se a exceção pode, de fato, ser lançada. Exemplo:

```
public class Calc {
   public int div(int a, int b) throws DivByZero {
      if (b = = 0) throw new DivByZero();
      return a/b;
   }
}
```

Quando uma exceção é lançada por um método, sua execução é interrompida e o controle volta para o método que o invocou. O método invocador deve então capturar a exceção

como descrito a seguir, ou relançar a exceção para que ela seja capturada mais alto na hierarquia das chamadas de métodos.

Captura de Exceções

Para capturar uma exceção é necessário utilizar a seguinte estrutura de código:

- ✓ O código que pode lançar a exceção deve ser inserido num bloco precedido da palavra-chave `try`. O interpretador Java tentará executar o bloco normalmente, a não ser que uma exceção seja lançada por um comando ou por um método executado dentro do bloco.
- ✓ Um tratador de exceções é implementado em Java através de um bloco precedido pela palavra-chave `catch` seguida pelo tipo de exceção em questão. O bloco `catch` responsável por tratar uma exceção lançada de um bloco `try` deve estar imediatamente após este último no texto do programa. Se vários tipos de exceção puderem ser lançados no bloco `try`, pode-se fornecer um bloco `catch` para cada tipo de exceção. Quando uma exceção de tipo `T` é lançada no bloco `try`, o controle é transferido para o bloco `catch` associado que aparecer primeiro lexicamente e que trate exceções do tipo `T` ou algum dos seus supertipos. Após a execução de um bloco `catch`, a execução do programa salta para o comando seguinte ao último bloco `catch` associado ao bloco `try` anterior.
- ✓ Também é possível associar a um bloco `try` um bloco precedido da palavra-chave `finally`, responsável por realizar ações de limpeza (Do inglês: *clean-up actions*). Blocos `finally` sempre são executados, indepentemente de uma exceção ter sido lançada no bloco `try` associado. Nos casos em que uma exceção é lançada, o bloco `finally` é executado imediatamente após a execução do bloco `catch` que trata a exceção, caso haja algum. Blocos `finally` aparecem depois de todos os tratadores associados a um bloco `try` ou imediatamente após este último, caso não existam tratadores.

Exemplo:

```
...
```

```
Calc calc = new Calc();
try {
   int div = 0;
   div = calc.div(x, y);
   System.out.println(div);
} catch (DivByZero dbz) {
   // Do something specific with the exception.
} catch (Exception e) {
   // Do something generic with the exception.
} finally {
   // Free resources and perform clean-up.
}
doSomething();
...
```

O exemplo acima ilustra uma situação em que existem dois blocos `catch`, além de um bloco `finally`. Neste caso, a execução dos blocos `catch` é totalmente dependente do tipo da exceção levantada no código do bloco `try`. Por exemplo, caso seja levantada a exceção verificada `DivByZero`, o primeiro bloco `catch` é o que será executado. Por outro lado, caso seja levantada uma exceção não-verificada, como `NullPointerException`, será executado o segundo bloco `catch`. Caso não seja levantada nenhuma exceção no bloco `try`, o fluxo de execução não passará por nenhum dos blocos `catch` e irá diretamente para o bloco `finally`. É importante ressaltar que o bloco `finally` será sempre executado, ou seja, não importa se alguma exceção é levantada no bloco `try`, nem qual é o tipo dessa exceção e nem qual é o código executado no bloco `catch`. Finalmente, se a execução do bloco `try-catch-finally` terminar normalmente (sem que exceções sejam lançadas a partir dos blocos `catch` e `finally`), o código executado em seguida será `doSomething()`.

2.2 Programação Orientada a Aspectos

Separação de interesses (Do inglês: *Separation of Concerns*) [20] é um princípio de Engenharia de Software que expõe a necessidade de manipular um tópico importante ou parte do problema de cada vez durante o projeto de um sistema. Um interesse pode ser entendido como uma parte específica do problema, à qual se deseja aplicar um tratamento diferenciado [8]. Interesses podem ser funcionais, como características ou regras de negócio pertinentes ao domínio do sistema, ou não-funcionais, como segurança e persistência. Em sistemas complexos, normalmente existem interesses que fazem parte de vários módulos e cujo código está misturado com o código de outros interesses. Esses interesses são conhecidos como **interesses transversais** (Do inglês: *crosscutting concerns*), pois eles aparecem de forma transversal a outros interesses no sistema. Alguns exemplos bem-conhecidos de interesses transversais são: *logging*, autenticação e tratamento de exceções.

Os modelos de programação tradicionais apresentam limitações ao tratar os interesses transversais. Em Programação Orientada a Objetos (POO), classes são um mecanismo efetivo para se modularizar a maioria dos interesses funcionais de um sistema, mas interesses transversais ficam espalhados por vários módulos em pequenos trechos de código que são, em geral, repetitivos, resultando em sistemas difíceis de projetar, entender, implementar, manter e evoluir.

A Programação Orientada a Aspectos (POA) [3, 8] foi proposta como uma técnica que permite a modularização de interesses transversais na construção de sistemas de software orientados a objetos, incrementando sua re-usabilidade e facilidade de evolução. Inicialmente proposta por um conjunto de pesquisadores do Xerox PARC [3], a Programação Orientada a Aspectos introduziu uma nova dimensão e um conjunto de diretrizes na Programação Orientada a Objetos para facilitar o desenvolvimento de software. POA possibilita a modularização de interesses transversais através do uso de uma nova abstração, denominada **aspecto** [43] (Do inglês: *aspect*), e de mecanismos específicos que tornam possível combinar vários aspectos em um programa coerente [8].

A Figura 4, extraída de [21], fornece uma visão geral das etapas do desenvolvimento de software orientado a aspectos. No passo (a) são identificados os interesses relativos aos

requisitos do sistema. No passo seguinte, os requisitos funcionais do sistema são implementados gerando um conjunto de componentes escritos em uma linguagem base ou de componentes, como Java [9]. Ainda no passo (b), um conjunto de aspectos relacionados às propriedades que afetam o comportamento do sistema e implementam requisitos não-funcionais são escritos usando uma linguagem orientada a aspectos, como AspectJ. No passo (c), os aspectos são combinados com o código escrito na linguagem base através de um processo conhecido como **combinação** [43] (Do inglês: *weaving*), que pode ser automaticamente executado por uma ferramenta da linguagem orientada a aspectos conhecida como **combinador** [43] (Do inglês: *weaver*).

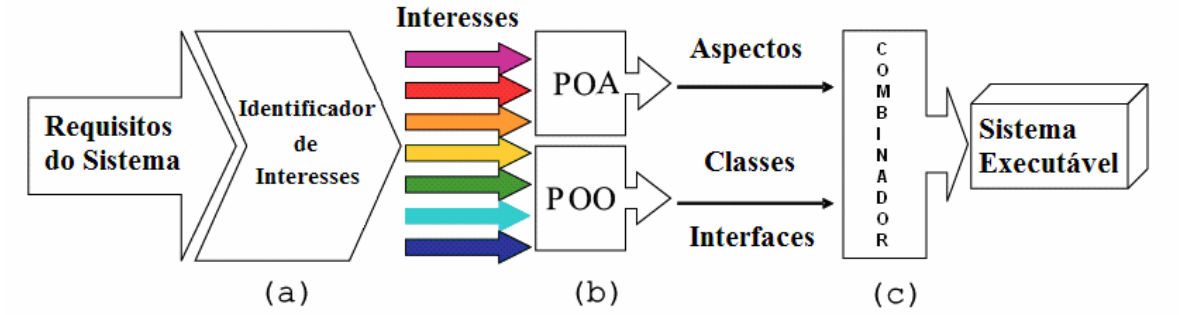

Figura 4 – Desenvolvimento Orientado a Aspectos [21]

2.2.1 AspectJ

AspectJ [4] é uma uma linguagem de propósito geral que estende a linguagem Java com os conceitos de POA. A linguagem define uma nova unidade de encapsulamento, chamada **aspecto** [43], que é similar a uma classe e, adicionalmente, pode incluir as novas construções definidas por AspectJ. Ela utiliza Java como linguagem base para a implementação dos interesses não-transversais e os aspectos para implementar os interesses transversais.

Esta seção descreve sucintamente a linguagem AspectJ, com ênfase nos mecanismos necessários ao entendimento deste trabalho. Foi considerada a versão 1.2 de AspectJ. Maiores detalhes sobre essa linguagem podem ser encontrados na literatura [10, 11, 12, 13]. Alguns conceitos da linguagem AspectJ que merecem destaque são discutidos a seguir.

Pontos de Junção e Conjuntos de Pontos de Junção

Pontos de junção [43] (Do inglês: *join points*) representam pontos bem definidos na execução de um programa. Exemplos de pontos de junção são: chamada a métodos, execução de métodos, acesso a atributos, tratamento de exceções e inicialização estática. **Conjuntos de pontos de junção** [43] (Do inglês: *pointcuts*) são um mecanismo lingüístico fornecido por AspectJ para selecionar conjuntos de pontos de junção. AspectJ dá suporte à seleção de vários tipos de pontos de junção:

- ✓ Chamada a método – quando um método é chamado, não incluindo as chamadas `super` em métodos não-estáticos
- ✓ Execução de método – quando o código do corpo do método é executado
- ✓ Chamada a construtor – quando um objeto é construído e o construtor inicial do objeto é chamado
- ✓ Execução de construtor – quando o código do corpo de um construtor é executado, após as chamadas de `super` ou `this` existentes no construtor
- ✓ Execução de inicializador estático – quando o inicializador estático de uma classe é executado
- ✓ Pré-inicialização de objeto – antes que o código de inicialização de objeto para uma classe seja executado. Compreende o intervalo de tempo entre o início do primeiro construtor chamado e o início do construtor pai
- ✓ Inicialização de objeto – quando o código de inicialização de objeto para uma classe é executado. Compreende o intervalo de tempo entre o retorno da chamada ao construtor pai e o retorno da chamada do primeiro construtor
- ✓ Referência a atributo – quando um atributo não-constante é referenciado
- ✓ Atualização de atributo – quando um atributo sofre atualização
- ✓ Execução de tratador – quando um tratador de exceções é executado
- ✓ Execução de **comportamento transversal** (Do inglês: *advice*) – quando o código do corpo de um comportamento transversal é executado

Conjuntos de pontos de junção permitem que informações contextuais relacionadas ao conjunto de pontos de junção selecionado sejam capturadas. É possível expor informações como: o objeto que está sendo executado, os argumentos de um método e, em alguns casos,

o objeto no qual um ponto de junção está sendo chamado. A Tabela 1 exibe exemplos de conjuntos de pontos de junção válidos.

Tabela 1 – Exemplos de conjuntos de pontos de junção

Conjunto de pontos de junção	**Descrição**
`call(* MyClass.myMethod* (String,..))`	Chamada a qualquer método cujo nome começa com "`myMethod`", definido por um tipo chamado `MyClass` e cujo primeiro argumento seja do tipo `String`
`call(MyClass+.new(..))`	Chamada a qualquer construtor de `MyClass` ou de suas subclasses (Subclasse é indicado pelo símbolo '+').
`execution(public * com.mycompany..*.*(..))`	Todos os métodos públicos em todas as classes em qualquer pacote que seja um sub-pacote de `com.mycompany.`
`execution(MyClass+.new(..))`	Execução de qualquer construtor de `MyClass` ou de suas subclasses.
`get(PrintStream System.out)`	Acesso de leitura ao atributo `out` do tipo `PrintStream` na classe `System`
`set(int MyClass.x)`	Acesso de escrita ao atributo `x` do tipo `int` em `MyClass`
`handler(CreditCard*)`	Execução de um bloco `catch` que trata uma exceção de um tipo cujo nome começa com `CreditCard`
`Staticinitialization(MyClass+)`	Execução de bloco estático de `MyClass` ou de suas subclasses
`withincode(* MyClass.myMethod(..))`	Qualquer conjunto de pontos de junção interno ao escopo léxico de qualquer método my`Method()` de `MyClass`
`cflow(call(* MyClass.myMethod(..))`	Todos os pontos de junção no fluxo de controle de chamadas a qualquer método `myMethod()` em `MyClass`, incluindo chamadas ao próprio método especificado
`cflowbelow(call(* MyClass.myMethod(..))`	Todos os pontos de junção no fluxo de controle de chamadas a qualquer método

	`myMethod()` em `MyClass`, excluindo chamadas ao próprio método especificado
`this(JComponent+)`	Todos os pontos de junção onde a expressão `this instanceof JComponent` é verdadeira
`target(MyClass)`	Todos os pontos de junção onde o objeto no qual o método é chamado é do tipo `MyClass`
`args(String,..,int)`	Todos os pontos de junção onde o primeiro argumento é do tipo `String` e o último argumento é do tipo `int`
`if(EventQueue.isDispatchThread())`	Todos os pontos de junção onde `EventQueue.isDispatchThread()` é verdadeiro

Operadores **||**, **&&,** e **!** podem ser utilizados com conjuntos de pontos de junção individuais para criar conjuntos de pontos de junção mais complexos. O exemplo a seguir designa um conjunto de pontos de junção que captura chamadas aos métodos `m1()` ou `m2()` definidos por um tipo chamado `MyClass` :

```
call(* MyClass.m1()) || call(* MyClass.m2())
```

No próximo exemplo é designado um conjunto de pontos de junção para chamadas ao método `m1()`, definido por um tipo chamado `MyClass`, e que estejam no fluxo de controle de chamadas ao método `m2()`, também definido por um tipo chamado `MyClass` :

```
call(* MyClass.m1()) && cflow(call(* MyClass.m2())
```

O exemplo abaixo ilustra um conjunto de pontos de junção para chamadas a todos os métodos, exceto ao método `m1()` definido por um tipo chamado `MyClass` :

```
!call(* MyClass.m1())
```

Comportamento Transversal

Comportamentos transversais (Do inglês: *advice*) especificam trechos de código que devem ser executados quando um conjunto de pontos de junção é atingido durante a execução do programa. Eles podem ser executados antes (*before*), depois (*after*), ou ao redor (*around*) dos pontos de junção selecionados. Um comportamento transversal *before* executa exatamente antes que um ponto de junção de interesse seja alcançado, enquanto um comportamento transversal *after* executa exatamente depois. Um comportamento transversal *around* contorna um ponto de junção e, além de poder adicionar comportamento tanto antes quanto depois do ponto de junção selecionado, pode substituir completamente a sua execução. Comportamentos transversais *around* também podem fazer com que a execução prossiga normalmente, mas com um conjunto de argumentos diferente.

O trecho de código abaixo define um conjunto de pontos de junção chamado `pc1()` que captura todas as chamadas a métodos públicos da classe `MyClass`. Logo em seguida é definido um comportamento transversal que está associado a `pc1()`. Como o comportamento transversal é do tipo *before*, o código dentro do corpo desse comportamento transversal será executado imediatamente antes de qualquer chamada a um método público de `MyClass`.

```
pointcut pc1() : call(public * MyClass.*(..));
before() : pc1() {
      System.out.println("Before:"+
System.currentTimeMillis());
}
```

Também é possível definir comportamentos transversais associados a conjuntos de pontos de junção anônimos, ou seja, que não foram definidos *a priori*. O trecho de código a seguir mostra como isso pode ser feito para o exemplo anterior:

```
before() : call(public * MyClass.*(..)) {
        System.out.println("Before:"+
System.currentTimeMillis());
```

```
}
```

AspectJ disponibiliza uma variável de referência especial, `thisJoinPoint`, que contém informação reflexiva sobre o ponto de junção. Esta variável é um objeto do tipo `org.aspectj.lang.JoinPoint` e só pode ser utilizada no contexto de um comportamento transversal. Abaixo, são exibidos alguns exemplos de comportamento transversal:

Neste exemplo, são exibidos o nome do método e o horário atual do sistema antes e após a chamada de qualquer método public na classe `MyClass`.

```
before() : call(public * MyClass.*(..)) {
       System.out.println("Before the method: "
           + thisJoinPoint.getSignature().getName() + "()  "
           + System.currentTimeMillis());
}
after() : call(public * MyClass.*(..)) {
       System.out.println("After the method: "
           + thisJoinPoint.getSignature().getName() + "()  "
           + System.currentTimeMillis());
}
```

No exemplo a seguir, todas as chamadas a `Connection.close()` são capturadas e, caso o *pool* de conexões esteja habilitado, a conexão é disponibilizada no *pool* em vez de ser fechada. Neste exemplo, a chamada ao método `Connection.close()` pode ser suprimida pelo comportamento transversal, caso o *pool* esteja habilitado.

```
void around(Connection conn) : call(Connection.close()) &&
target(conn) {
    if (enablePooling) {
        connectionPool.put(conn);
    } else {
```

```
        proceed(conn);
    }
}
```

Interesse Transversal Estático

Interesse transversal estático [43] (Do inglês: *static crosscutting*) está relacionado com a capacidade da linguagem AspectJ de, além de modificar o comportamento dinâmico do programa, permitir mudanças também na sua estrutura estática. São permitidas alterações como a adição de novos métodos e atributos, mudanças na hierarquia de tipos e substituição de exceções verificadas (Do inglês: *checked*) por não-verificadas (Do inglês: *unchecked*), descritas a seguir:

- **Inclusão de novos membros -** AspectJ permite que novos membros sejam adicionados a classes ou interfaces. É possível introduzir novos atributos, construtores, métodos abstratos ou concretos. No caso de interfaces, membros introduzidos por AspectJ passam a fazer parte de todas as classes que implementam a interface. O exemplo a seguir ilustra o código de um aspecto que introduz o método `foo()` e o atributo estático `instanceCount` do tipo `int` na classe `MyClass`:

```
aspect IntroduceMethodIllustration {
        private void MyClass.foo() {
                System.out.println("This is foo");
        }
        private static int MyClass.instanceCount = 0;
}
```

- **Reestruturação de hierarquia de tipos** - É possível modificar a hierarquia de herança das classes. Através de construções do AspectJ, podem ser declaradas super classes ou interfaces para uma classe ou interface existente. O exemplo a seguir declara que a classe `MyClass` é uma implementação da interface `Serializable`:

```
aspect MakeMyClassSerializable {
        declare parents : MyClass implements
Serializable;
}
```

- **Conversão de exceções verificadas em não-verificadas** - AspectJ pode transformar uma exceção verificada em não-verificada em um conjunto de pontos de junção de interesse. Exceções verificadas lançadas nesses pontos de junção são encapsuladas por `org.aspectj.lang.SoftException` (exceção não-verificada). Através desse artifício, torna-se possível ignorar as checagens estáticas que o compilador Java faz para exceções verificadas. No exemplo a seguir, métodos que fazem chamadas ao método `read()` da classe `FileInputStream` não precisam tratar a exceção verificada `IOException` que este último levanta. Quando a exceção `IOException` for levantada, será necessário tratar a exceção não-verificada `SoftException` que será lançada em seu lugar.

```
aspect SoftenReadIOException {
    declare soft : IOException :
        call(* FileInputStream.read());
}
```

Aspectos

Aspectos representam unidades de modularidade para os interesses transversais. Por serem similares às classes, eles também possuem um tipo, podem ser estendidos, podem ser abstratos ou concretos e podem conter campos, métodos e tipos como membros, embora não possam ser instanciados da mesma forma que classes. Todas as construções de Java que podem aparecer em uma classe também podem aparecer em um aspecto. Além disso, também podem aparecer em aspectos as novas construções introduzidas por AspectJ: **conjuntos de pontos de junção**, **comportamento transversal** e **interesse transversal estático**. Aspectos são combinados com classes Java através do processo de **combinação** e

o código resultante desse processo segue o padrão *bytecode* Java, que pode ser executado em qualquer máquina virtual Java.

O exemplo a seguir ilustra um aspecto que insere no código “boas maneiras” ao estilo japonês, adicionando o sufixo “-san” aos nomes de pessoas:

```
public aspect JapaneseMannersAspect {
    pointcut callSayMessageToPerson(String person)
        : call(* HelloWorld.sayToPerson(String, String))
              && args(*, person);
    void around(String person)
        : callSayMessageToPerson(person) {
        proceed(person + "-san");
    }
}
```

2.3 Conjunto de Métricas

Conforme mencionado na Seção 2.2, POA é uma técnica que permite separação de interesses na construção de sistemas de software orientados a objetos. O uso de POA para modularizar interesses impacta diversos atributos de qualidade de um sistema, como acoplamento, coesão e tamanho. Esse impacto pode ser positivo ou negativo, dependendo de diversos fatores como a complexidade dos interesses modularizados, o quão transversais eles são e a interação entre esses interesses.

Métricas de software são largamente utilizadas em estudos empíricos para verificar critérios de qualidade de um determinado sistema. Neste trabalho, optamos por usar um conjunto de métricas [1] capaz de avaliar implementações orientadas a aspectos a fim de quantificar o impacto na qualidade do sistema ao utilizar aspectos para separar o código responsável pelo tratamento de exceções do código normal do sistema. O conjunto de métricas quantifica atributos internos a um sistema, como separação de interesses, acoplamento, coesão e tamanho. Esses atributos internos são mais concretos, bem definidos e fáceis de medir e,

portanto, podem ser utilizados por um modelo de qualidade [1] como o que aparece na Figura 5. Os atributos internos são usados pelo modelo de qualidade para avaliar os atributos externos do sistema, como manutenibilidade e reusabilidade, os quais são de interesse direto do usuário [18]. Este conjunto de métricas já foi utilizado em alguns estudos experimentais [5, 6, 7, 16] e tem se mostrado eficiente para avaliar atributos de qualidade em programas Java e AspectJ. Além disso, é possível realizar a coleta de algumas dessas métricas de forma automática, utilizando-se ferramentas construídas especialmente com esse propósito [15].

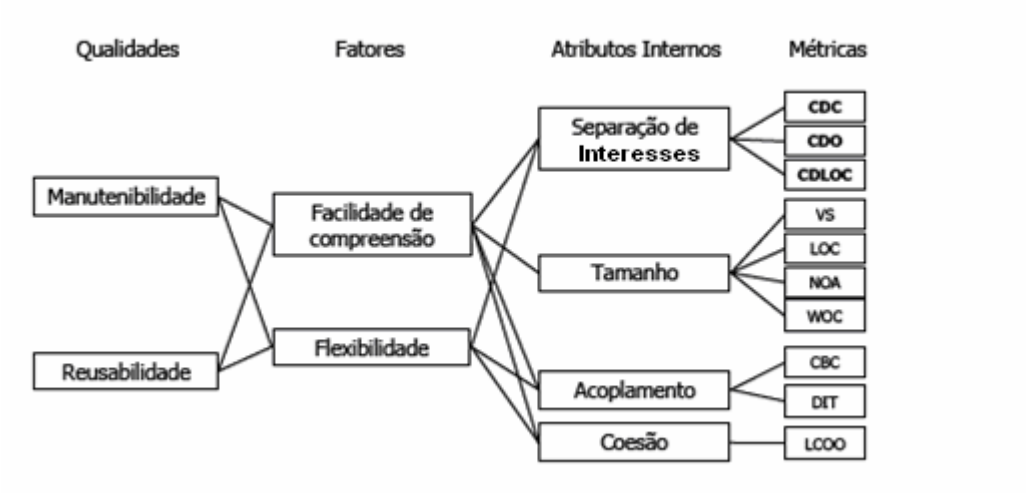

Figura 5 – O Modelo de Qualidade [1]

2.3.1 Métricas de Separação de Interesses

Separação de interesses (*Separation of Concerns - SoC)* [8], como já foi citado neste trabalho, é um princípio de Engenharia de Software que expõe a necessidade de manipular um tópico importante ou parte do problema de cada vez durante o projeto de um sistema. Refere-se à habilidade de identificar, encapsular e manipular as partes do sistema relacionadas a um interesse em particular, como persistência e distribuição [8]. As métricas de SoC utilizadas neste trabalho são: Difusão de Interesse por Componentes (Do inglês: *Concern Diffusion over Components* - CDC), Difusão de Interesse por Operações (Do inglês: *Concern Diffusion over Operations* - CDO) e Difusão de Interesse por Linhas de

Código (Do inglês: *Concern Diffusion over Lines of Code* - CDLOC). Essas métricas são descritas a seguir.

Difusão de Interesse por Componentes (CDC)

CDC contabiliza o número de components (classes e aspectos) que implementam um determinado interesse. Entra na conta qualquer classe que tenha código que pertence à implementação do interesse, mesmo que esse código seja uma parte pequena da implementação do componente. Além disso, ela conta o número de componentes que acessam esses componentes, por exemplo, usando-os em declarações de atributos, como parâmetros ou tipos de retorno de método, em declarações `throws`, em variáveis locais ou chamadas a seus métodos.

Difusão de Interesse por Operações (CDO)

CDO contabiliza o número de operações, isto é, métodos, construtores e comportamentos transversais dos components (classes e aspectos) que implementam um interesse em particular. Além disso, ela conta o número de operações que acessam esses componentes, por exemplo, usando-os em declarações de atributos, como parâmetros ou tipos de retorno de método, em declarações `throws`, em variáveis locais ou chamadas a seus métodos.

Difusão de Interesse por Linhas de Código (CDLOC)

CDLOC contabiliza o número de pontos de transição para cada interesse através das linhas de código. Essa métrica avalia o quão misturado e espalhado está um determinado interesse no sistema. Quanto maior for o valor de CDLOC, mais misturado o interesse está no código que implementa os componentes; quanto menor for o CDLOC, mais localizado está o interesse no código. Maiores detalhes de como essa métrica é calculada podem ser vistos em [17].

2.3.2 Métricas de Acoplamento

Acoplamento é uma indicação da força das interconexões entre os componentes de um sistema [19]. Sistemas com alto acoplamento têm interconexões fortes, com unidades do programa dependentes entre si [19], o que implica em um programa mais difícil de modificar e entender. Neste trabalho são utilizadas duas métricas de acoplamento: Acoplamento entre Componentes (Do inglês: *Coupling between Components* - CBC) e Profundidade da Árvore de Herança (Do inglês: *Depth of Inheritance Tree* - DIT). Essas métricas são explicadas a seguir.

Acoplamento entre Componentes (CBC)

CBC é definida para um componente (classe ou aspecto) como o número de outros componentes aos quais ele está acoplado. Para cada componente, conta-se o número de outras classes que são usadas em declarações de atributos, parâmetros de operações, tipos de retorno ou variáveis locais. Além disso, para cada aspecto, conta-se: (i) as classes nas quais são introduzidos atributos e métodos através do interesse transversal estático, as classes ou aspectos que são interceptados pelo aspecto por meio de definições de conjuntos de pontos de junção; e (ii) os componentes cujos atributos e métodos introduzidos via interesse transversal estático são acessados pelo aspecto. Se um componente X se relaciona a um mesmo componente Y várias vezes da mesma forma ou de formas diferentes de acoplamento, o valor de CBC do componente X é incrementado em apenas um, pois essa métrica conta o número de componentes com o qual determinado componente está acoplado e não o número de vezes com que esse acoplamento se dá.

Profundidade da Árvore de Herança (DIT)

DIT é definida como o comprimento máximo de um nó para a raiz da árvore. Ela mede a profundidade na hierarquia de herança em que um componente é declarado. Convenciona-se que classes que herdam diretamente de `Object` e interfaces têm DIT zero.

2.3.3 Métricas de Coesão

Coesão de um componente é uma medida que avalia o quanto seus elementos internos estão fortemente relacionados entre si [19]. Quanto maior a coesão dos componentes de um sistema, maior a sua qualidade. Neste trabalho é usada a métrica de coesão Falta de Coesão nas Operações (Do inglês: *Lack of Cohesion in Operations* - LCOO). Essa métrica é detalhada a seguir.

A Falta de Coesão nas Operações (LCOO)

LCOO mede a falta de coesão de um componente (classe ou aspecto). Se um componente *C1* tem *n* operações (métodos e comportamentos transversais) *O1,..., On*, então *{Ij}* é o conjunto de atributos usados pela operação Oj. Seja |*P*| o número de interseções vazias entre conjuntos de atributos. Seja |*Q*| o número de interseções não vazias entre conjuntos de atributos. Então: LCOO = |*P*| – |*Q*|, *se* |*P*| > |*Q*|, ou LCOO = *0*, se |*P*| < |*Q*|. LCOO mede a quantidade de pares de métodos/comportamentos transversais que não acessam pelo menos um mesmo atributo. A intuição por trás dessa métrica é que se *C1* tem conjuntos *OS1, OS2, ..., OSn* de operações que acessam conjuntos disjuntos *I1, I2, ..., In* de atributos, talvez *OS1, OS2, ..., OSn* e os respectivos conjuntos de atributos não devessem estar no mesmo componente. LCOO leva em consideração, além dos comportamentos transversais e métodos internos ao aspecto, também os métodos que o aspecto introduz nas classes via interesse transversal estático.

2.3.4 Métricas de Tamanho

As métricas de tamanho tratam de diferentes aspectos do tamanho do sistema. Neste trabalho são usadas quatro métricas de tamanho: Tamanho do Vocabulário (Do inglês: *Vocabulary Size* - VS), Linhas de Código (Do inglês: *Lines of Code* - LOC), Número de Atributos (Do inglês: *Number of Attributes* - NOA) e Número de Operações (Do inglês: *Number of Operations* - NOO). Essas métricas são descritas a seguir.

Tamanho do Vocabulário (VS)

VS conta o número de componentes do sistema, isto é, o número de classes, interfaces e aspectos que constituem o sistema. Essa métrica mede o tamanho do vocabulário do sistema, onde cada nome de componente é contado como parte do vocabulário do sistema. O raciocínio por trás da métrica é que, quanto maior o tamanho do vocabulário do sistema, maior a quantidade de conceitos que alguém sem familiaridade com o sistema precisará entender para modificá-lo.

Linhas de Código (LOC)

LOC conta o número de linhas de código. É uma medida de tamanho tradicional. Comentários relativos à documentação ou implementação não são interpretados como código. Linhas em branco também não são contadas. As ferramentas adotadas neste trabalho para captura automática [15] dessa métrica utilizam alguns critérios, além dos citados acima, para que estilos diferentes de programação não influenciem os resultados.

Número de Atributos (NOA)

NOA mede o vocabulário interno de cada componente, isto é, o número de atributos de cada classe ou aspecto. Atributos herdados de superclasses ou superaspectos não são contados por essa métrica. Num aspecto, além dos atributos internos ao aspecto, são contados também os atributos que o aspecto introduz nas classes via interesse transversal estático.

Número de Operações (NOO)

NOO mede a complexidade de um componente em termos de suas operações (métodos e comportamentos transversais). Essa métrica conta o número de operações em um determinado componente. Para as classes, conta-se os métodos e construtores. Em um aspecto, além dos comportamentos transversais e métodos internos ao aspecto, são considerados também os métodos que o aspecto introduz nas classes via interesse transversal estático.

Capítulo 3

TRATAMENTO DE EXCEÇÕES COM POA EM SISTEMAS ORIENTADOS A OBJETOS

Este capítulo detalha dois estudos de caso que avaliam as vantagens e desvantagens da programação orientada a aspectos para modularizar o código de tratamento de exceções em sistemas orientados a objetos que não usam originalmente POA em sua estruturação. Para medir os resultados desses estudos foi utilizado o conjunto de métricas mostrado na Seção 2.3. As métricas foram coletadas para as versões originais (OO) e refatoradas (OA) dos sistemas em questão.

A refatoração realizada nesses estudos de caso corresponde à "aspectização" do código de tratamento de exceções, isto é, ao processo de utilizar aspectos para separar o código responsável pelo tratamento de exceções do código normal do sistema. Usamos o termo "tratador de exceções" para designar blocos *try-catch*, *try-catch-finally* e *try-finally* (Seção 2.1). A linguagem orientada a aspectos (OA) utilizada nos estudos de caso foi AspectJ. Nos dois sistemas, a sistemática utilizada para a refatoração foi a seguinte: para cada tratador de exceções existente na versão original, foi criado um comportamento transversal em um aspecto (Seção 2.2). Em situações em que foi possível haver reuso de código, um único comportamento transversal foi utilizado para implementar vários tratadores de exceções existentes no código original.

Este capítulo está dividido em quatro seções. Na Seção 3.1 são descritos os sistemas usados nos estudos de caso deste capítulo. A Seção 3.2 é dedicada à exposição dos resultados obtidos para as versões originais e refatoradas dos dois sistemas. Finalmente, na Seção 3.3 são apresentadas as conclusões obtidas com relação à análise dos resultados dos dois estudos de caso.

3.1 Descrição dos Sistemas Usados

Nesta seção são detalhados o sistema *Java Pet Store* usado no primeiro estudo de caso e o sistema Telestrada usado no segundo estudo de caso. No primeiro estudo de caso, a refatoração do sistema *Java Pet Store* e a coleta de métricas foram executadas no decorrer do desenvolvimento deste trabalho. Já no segundo estudo de caso, o sistema Telestrada foi refatorado por Fernando Castor Filho e a coleta de métricas foi realizada como parte do escopo deste trabalho, seguindo o mesmo processo utilizado nos outros dois estudos de caso.

3.1.1 O Sistema *Java Pet Store*

O sistema usado no primeiro estudo de caso deste trabalho é denominado **Java™ Pet Store Demo**. Ele corresponde a um exemplo de sistema para a plataforma Java 2 Enterprise Edition (J2EE) disponibilizado pela Sun Microsystems [9]. O *Java Pet Store* funciona como uma loja virtual que vende animais pela *Internet* e é um exemplo típico de sistema de comércio eletrônico. O sistema é baseado em diversas tecnologias da plataforma J2EE usadas com freqüência na construção de sistemas de informação voltados para a *web*, por exemplo, Enterprise Javabeans [28], Servlets [29], Java Server Pages [30], Java Messaging Service [31] e JavaMail [32].

O sistema possui um *Web site* que disponibiliza uma interface com os clientes. Existem também outras interfaces que são usadas por parceiros comerciais, como os fornecedores, e por administradores da loja para realizar atividades administrativas e de controle de estoque. As funcionalidades do sistema são agrupadas em categorias e apresentadas aos clientes seguindo as regras de controle de acesso. O sistema realiza a maioria das tarefas automaticamente, mas algumas tarefas são feitas manualmente, por exemplo, gerenciamento de estoque e de pedidos. A Figura 6, baseada em [24], fornece uma visão geral das regras de negócio desse sistema.

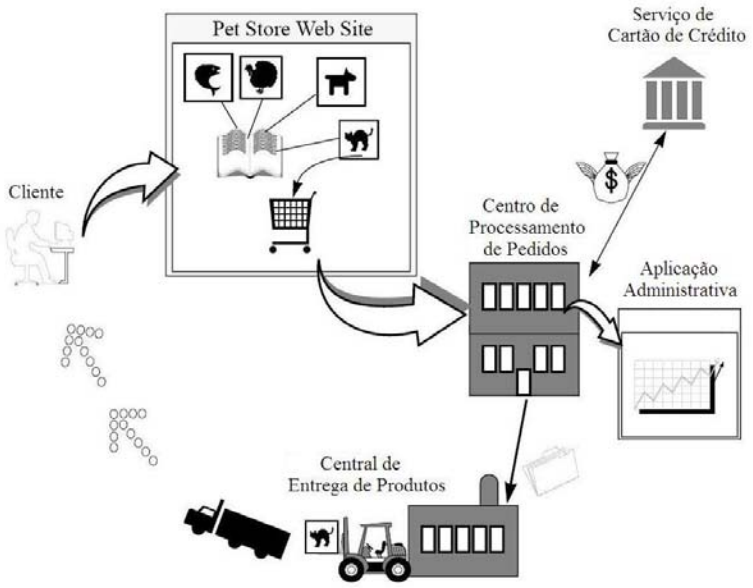

Figura 6 – Regras de Negócio do Sistema *Java Pet Store*

O sistema divide-se em quatro sub-aplicações distintas:

- ✓ ***Web Site* de Comércio Eletrônico** ("`petstore`") – aplicação utilizada pelos usuários da *Internet* para comprar os produtos. Permite que o cliente compre produtos, gerando pedidos de compra que são enviados à Central de Processamento de Pedidos.
- ✓ **Aplicação Administrativa** ("`petstoreadmin`") – aplicação utilizada pelos administradores da empresa para visualizar estatísticas de vendas e, manualmente, aceitar ou rejeitar pedidos.
- ✓ **Central de Processamento de Pedidos** ("`OPC`") – aplicação orientada a processos que controla o processamento de pedidos e as transações financeiras. Alguns exemplos de funcionalidades dessa aplicação são: (i) recebimento e processamento de pedidos criados na aplicação "`petstore`"; (ii) envio de *email* para os clientes confirmando os pedidos realizados; e (iii) envio de pedidos para os fornecedores.
- ✓ **Central de Entrega de Produtos** ("`supplier`") – aplicação orientada a processos que controla o abastecimento e a entrega de produtos aos clientes. Alguns exemplos de funcionalidades dessa aplicação são: (i) recebimento de pedidos enviados pela aplicação "`OPC`"; e (ii) despacho dos produtos para os clientes.

Mais detalhes sobre esse sistema podem ser obtidos em um outro documento [24].

A implementação total do *Java Pet Store* contém em torno de 17500 linhas de código (LOC) e 330 classes e interfaces. O código de tratamento de exceções original das classes possui aproximadamente 300 tratadores de exceções de complexidade variada, que variam desde um bloco *catch* vazio até trechos complexos de código contendo informação contextual das classes.

3.1.2 O Sistema Telestrada

O Telestrada é um sistema real desenvolvido com o intuito de fornecer aos usuários das rodovias federais brasileiras informações diversas, como por exemplo, as condições destas rodovias. Ele também é responsável por coletar sugestões e reclamações sobre os serviços prestados pelas operadoras que administram essas rodovias, visando estabelecer um canal de comunicação entre usuários e o governo federal para assuntos relacionados à operação e conservação dessas vias.

O sistema armazena inúmeras informações sobre as rodovias, que vão desde as condições das estradas até as facilidades existentes ao longo dos trechos, por exemplo, a localização de postos de combustível.

O Telestrada possui cinco módulos básicos:

- ✓ **Módulo de Cadastro Rodoviário** – responsável pela inserção, consulta e alteração de dados rodoviários;
- ✓ **Módulo de Eventos** – destinado ao registro de ocorrências na malha rodoviária que devam ser observadas pelos usuários;
- ✓ **Módulo de Consultas de Rotas e Condições de Rodovias** – permite consulta de informações sobre trechos da malha rodoviária;
- ✓ **Módulo de Emergências** – destinado à solicitação de atendimento de emergência em trechos da malha rodoviária;
- ✓ **Módulo de Reclamações** – responsável pelo registro e consulta de reclamações sobre os serviços prestados pelas operadoras que administram malhas rodoviárias.

Neste estudo de caso foi selecionada parte do Módulo de Reclamações. Este módulo define operações para registrar, atualizar e consultar processos de reclamação de trechos da malha

rodoviária. Através deste módulo, um usuário da rodovia pode registrar reclamações pela *web* e consultar o andamento de um processo de reclamação. Além destas operações, o módulo de reclamação também define operações privadas a administradores do sistema (supervisores e responsáveis por trechos) que tornam possível registrar e remover tipos de reclamação e definir, encaminhar e finalizar um processo de reclamação.

A implementação total do Módulo de Reclamações contém em torno de 12000 LOC de código Java e 300 classes e interfaces. Os pacotes selecionados por este estudo de caso compreendem aproximadamente 3350 LOC de código Java, excluindo-se comentários e linhas em branco, e 220 classes e interfaces.

Analogamente ao estudo de caso do sistema *Java Pet Store* (Seção 3.1), neste estudo de caso a refatoração do sistema também tem como objetivo mover o código destinado a tratamento de exceções das classes/interfaces para aspectos criados especificamente com este propósito. O código original para tratamento de exceções nas classes compreende em torno de 50 blocos *try-catch, try-finally* e *try-catch-finally*. Esses blocos implementam diversas estratégias de tratamento de exceções que variam desde um bloco *catch* vazio até trechos complexos de código utilizando API Java de reflexão para criação dinâmica de métodos.

3.2 Resultados

Finalizada a etapa de refatoração dos sistemas *Java Pet Store* e Telestrada, utilizou-se o conjunto de métricas definido na Seção 2.3 para analisar quantitativamente as versões originais e refatoradas destes sistemas. Nesta seção são apresentados os valores obtidos para as métricas. Para facilitar a comparação dos resultados, as medidas coletadas para as versões OO (original) e OA (refatorada) dos dois sistemas são exibidas em tabelas. No restante deste capítulo, exceto onde for explicitamente indicado, o termo **classe** é utilizado para designar tanto classes como interfaces e o termo **componente** designa tanto aspectos quanto classes. A coleta dos valores para as métricas de separação de interesses foi feita manualmente. Com exceção da métrica Número de Atributos que foi coletada com a

ferramenta CASE Together 6.3 [25], as métricas restantes foram coletadas com a ferramenta AopMetrics [15].

3.2.1 Métricas de Separação de Interesses

A Tabela 2 exibe os resultados obtidos para as métricas de separação de interesses calculadas para o sistema *Java Pet Store*, utilizado no primeiro estudo de caso. A versão OA do *Java Pet Store* obteve melhores resultados para as três métricas de separação de interesses. A estratégia utilizada para refatoração determinou que todo o código relacionado à implementação do interesse de tratamento de exceções fosse movido para aspectos. Porém, houve uma situação na qual o código permaneceu inalterado: trechos de código para tratamento de exceções que foram criados automaticamente por um gerador de código. Por este motivo, os valores dessas métricas para algumas classes na versão OA são diferentes de zero.

Na métrica Difusão de Interesse por Componentes, o valor obtido para a versão OA do *Java Pet Store* foi aproximadamente 50% menor do que o valor obtido para a versão OO. É importante ressaltar que esse valor depende diretamente da abordagem selecionada para a criação de aspectos. Neste estudo de caso, optou-se por criar um aspecto dedicado a tratamento de exceções para cada pacote contendo classes implementando tratamento de exceções. Outras abordagens possíveis são a criação de um aspecto para cada classe, a criação de um aspecto por exceção, ou mesmo a criação de um único aspecto. A abordagem escolhida foi um meio-termo entre as citadas acima, isto é, entre a criação de um único aspecto excessivamente grande ou várias dezenas de aspectos minúsculos.

Na métrica Difusão de Interesse por Operações, a versão OA também obteve resultados melhores do que os obtidos para a versão OO, apresentando uma diminuição de aproximadamente 20% com relação à versão OO. Na versão OA, todas as classes obtiveram valores menores ou iguais aos valores obtidos para a versão OO.

Difusão de Interesse por Linhas de Código foi a métrica para a qual a versão OA do *Java Pet Store* apresentou os melhores resultados, entre todas as métricas coletadas. A versão OA chegou a uma redução de aproximadamente 90% com relação à versão OO e este percentual não atingiu 100%, como foi explicado anteriormente, devido à decisão de não

refatorar o código de tratamento de exceções em classes que foram geradas de maneira automática.

Tabela 2 – Métricas de Separação de Interesses para o sistema *Java Pet Store*

Métricas	Componentes	Versão OO	Versão OA
Difusão de Interesse por Componentes	Classes	110	20
	Aspectos	-	37
	Total	**110**	**57**
	Diferença	**-48,18%**	
Difusão de Interesse por Operações	Classes	256	21
	Aspectos	-	179
	Total	**256**	**200**
	Diferença	**-21,88%**	
Difusão de Interesse por Linhas de Código	Classes	1168	84
	Aspectos	-	0
	Total	**1168**	**84**
	Diferença	**-92,81%**	

Na Tabela 3 são apresentados os valores coletados para as métricas de separação de interesses no sistema Telestrada. A versão OA do Telestrada foi mais bem sucedida do que a versão OO, apresentando resultados melhores para duas das três métricas de separação de interesses. Neste estudo de caso, a estratégia utilizada para refatoração foi a mesma utilizada no primeiro estudo de caso: o código relacionado à implementação do interesse de tratamento de exceções existente nas classes foi movido para aspectos. Como foi possível mover todo o código de tratamento de exceções das classes para os aspectos, os valores obtidos nas métricas de separação de interesse para todas as classes da versão OA do Telestrada são iguais a zero.

Na métrica Difusão de Interesse por Componentes, a versão OA do Telestrada apresentou uma diminuição de aproximadamente 18% com relação à versão OO. Como foi dito anteriormente, esse valor depende diretamente da abordagem selecionada para a criação de aspectos. Neste estudo de caso, os aspectos foram criados de acordo com a complexidade dos cenários encontrados, isto é, com base na quantidade de código responsável por tratar exceções em cada classe. Um aspecto tratador foi criado para cada classe com um grande número (10+) de tratadores. Além disso, para cada pacote foi criado um aspecto

responsável por modularizar o tratamento de exceções em classes mais simples (com menos código de tratamento de exceções).

Tabela 3 – Métricas de Separação de Interesses para o sistema Telestrada

Métricas	Componentes	Versão OO	Versão OA
Difusão de Interesse por Componentes	Classes	22	0
	Aspectos	-	18
	Total	**22**	**18**
	Diferença	**-18,18%**	
Difusão de Interesse por Operações	Classes	42	0
	Aspectos	-	44
	Total	**42**	**44**
	Diferença	**+4,76%**	
Difusão de Interesse por Linhas de Código	Classes	208	0
	Aspectos	-	0
	Total	**208**	**0**
	Diferença	**-100%**	

Para a métrica Difusão de Interesse por Operações, a versão OA do Telestrada surpreendentemente apresentou resultados piores que os obtidos para a versão OO, com um aumento de aproximadamente 5% no valor medido. A maioria dos componentes na versão OA apresentou resultados melhores, porém algumas exceções foram encontradas, como a classe `db.GenericOperations`, para a qual a versão OA obteve resultados piores. Ao analisar esse caso, percebeu-se que o motivo do aumento na métrica foi o fato de que, na versão OO, alguns métodos possuíam mais de um bloco `try-catch` e para movê-los para aspectos foi necessário colocá-los em comportamentos transversais separados. Em cenários como esse, o valor obtido para CDO foi igual a 1 na versão OO e maior ou igual a 2 na versão OA. Segue um exemplo deste cenário encontrado no sistema Telestrada:

```
/* Classe GenericOperations */
public class GenericOperations {
      . . .
   public GenericOperations() throws SQLException, PoolException,
      Exception {
    preparedStatementList = new Vector();
    System.out.println(" - Vou fazer getInstance...");
    try {
      ResourceBundle bundle = ResourceBundle.getBundle("connectionpool");
      ConnectionPool pool = ConnectionPool.getInstance(DB);
      System.out.println(" - Vou fazer pool.get()...");
```

```
      connection = (Connection) pool.get();
      if (connection != null) {
        System.out.println(" - conn != null");
        try {
          connection.setAutoCommit(false);
        } catch (SQLException e) {
          System.out.println(" - Could not set auto-commit = false." +
            e.toString());
          close();
          throw e;
        }
      } else {
        System.out.println(" - conn == null");
        System.out.println(" - Unable to get a connection from the
            connection pool.");
        throw new Exception("Unable to get a connection from the
            connection pool.");
      }
    } catch (PoolException e) {
      System.out.println(" - Error getting a database connection from"+
            " ConnectionPool object" + e.toString());
      throw e;
    } catch (Exception e) {
      System.out.println(" - Error " + e.toString());
      throw e;
    }
  }
      . . .
}
```

Segue a implementação OA do mesmo trecho de código exibido acima:

```
/* Classe GenericOperations refatorada */
public class GenericOperations {
      . . .
   public GenericOperations() throws SQLException, PoolException,
      Exception {
    preparedStatementList = new Vector();
    System.out.println(" - Vou fazer getInstance...");
    ResourceBundle bundle = ResourceBundle.getBundle("connectionpool");
    ConnectionPool pool = ConnectionPool.getInstance(DB);
    System.out.println(" - Vou fazer pool.get()...");
    connection = (Connection) pool.get();
    if (connection != null) {
      System.out.println(" - conn != null");
      connection.setAutoCommit(false);
    } else {
      System.out.println(" - conn == null");
      System.out.println(" - Unable to get a connection from the
            connection pool.");
      throw new Exception("Unable to get a connection from the
            connection pool.");
    }
  }
```

```
      . . .
}
/* Aspecto GenericOperationsHandler */
privileged public aspect GenericOperationsHandler {
      . . .
  pointcut GenericOperationsConstructorHandler_1() :
      execution(GenericOperations.new());
  pointcut setManualCommitHandler() : call(*
      Connection.setAutoCommit(..)) &&
      withincode(GenericOperations.new());
  after(GenericOperations go) throwing(SQLException e) throws
      SQLException : setManualCommitHandler() && this(go){
    go.close();
  }
  after(GenericOperations go) throwing (Exception e) :
      GenericOperationsConstructorHandler_1() && this(go) {
    if (e instanceof PoolException) {
      System.out.println(" - Error getting a database connection from"+
            " ConnectionPool object" + e.toString());
    } else {
      System.out.println(" - Error " + e.toString());
    }
  }
      . . .
}
```

Analogamente aos resultados obtidos para o sistema *Java Pet Store*, a Difusão de Interesse por Linhas de Código para o sistema Telestrada foi a métrica que obteve os melhores resultados na versão OA dentre todas as métricas coletadas. A versão OA chegou a uma redução de 100% com relação à versão OO, o que representa uma separação textual total entre o código que trata exceções e o código responsável pelos outros interesses do sistema.

3.2.2 Métricas de Acoplamento e Coesão

A Tabela 4 exibe os resultados obtidos para as métricas de acoplamento e coesão para o sistema *Java Pet Store*. As versões OO e OA desse sistema obtiveram valores bastante próximos para as duas métricas de acoplamento.

A métrica Acoplamento entre Componentes sofreu um aumento aproximado de 1.5% na versão OA. Observou-se que o aumento dessa métrica está diretamente relacionado à ocorrência de cenários em que os tratadores de exceções precisam capturar informação contextual das classes. A métrica Profundidade da Árvore de Herança também piorou na versão OA, sofrendo um aumento de aproximadamente 5%. O aumento de 13 no valor da

medida corresponde à criação de 4 aspectos abstratos que foram estendidos por 13 aspectos. Os aspectos abstratos foram criados com o intuito de permitir reaproveitamento de código entre aspectos.

Falta de Coesão nas Operações foi a métrica que apresentou os piores resultados na versão OA do *Java Pet Store*, apresentando um aumento de aproximadamente 8% na versão OA. Apesar de não ter atingido a maior diferença percentual entre todas as métricas, ela destacou-se por não ter apresentado melhoria para nenhum componente na versão OA. Na maioria dos casos as duas versões apresentaram resultados equivalentes. Os casos em que os componentes apresentaram-se piores na versão OA puderam ser enquadrados em dois cenários principais: classes para as quais foram extraídos métodos para expor pontos de junção e aspectos possuindo atributos.

Tabela 4 – Métricas de Acoplamento e Coesão para o sistema *Java Pet Store*

Métricas	Componentes	Versão OO	Versão OA
Acoplamento entre Componentes	Classes	783	729
	Aspectos	-	65
	Total	**783**	**794**
	Diferença	**+1,4%**	
Profundidade da Árvore de Herança	Classes	245	245
	Aspectos	-	13
	Total	**245**	**258**
	Diferença	**+5,31%**	
Falta de Coesão nas Operações	Classes	7095	7595
	Aspectos	-	71
	Total	**7095**	**7666**
	Diferença	**+8,05%**	

A Tabela 5 apresenta os valores obtidos para as métricas de acoplamento e coesão calculadas para o sistema Telestrada. Semelhante ao observado para o sistema *Java Pet Store*, as versões OO e OA do Telestrada também obtiveram valores próximos para as métricas de acoplamento.

A métrica Acoplamento entre Componentes sofreu um aumento aproximado de 1.1% na versão OA. Assim como aconteceu na versão OA do *Java Pet Store*, o pequeno aumento no valor dessa métrica na versão OA do Telestrada teve como causa a ocorrência de cenários onde aspectos tratadores precisam capturar informação contextual relativa ao lançamento

de uma exceção. O valor obtido para a métrica Profundidade da Árvore de Herança também mudou pouco na versão OA, sofrendo um aumento de aproximadamente 1%. Neste caso, o aumento de 2 no valor da medida corresponde à criação de 1 aspecto abstrato que foi estendido por 2 aspectos.

Semelhante ao ocorrido no *Java Pet Store*, o resultado obtido para a métrica Falta de Coesão nas Operações também foi o pior entre todas as métricas na versão OA do Telestrada, apresentando um aumento de aproximadamente 28%. Neste caso, além de indicar que não houve melhoria para nenhum componente na versão OA, ela também foi a métrica com maior aumento percentual. De forma análoga ao observado no *Java Pet Store*, classes em que foram criados novos métodos para expor pontos de junção de interesse foram as principais responsáveis pelo resultado ruim. Porém, o segundo cenário, aspectos possuindo atributos, não foi identificado no Telestrada. Conforme mostrado na Tabela 5, os aspectos na versão OA do sistema não contribuíram para o valor total da métrica, já que nenhum dos comportamentos transversais acessa atributos dos aspectos ou das classes.

Tabela 5 – Métricas de Acoplamento e Coesão para o sistema Telestrada

Métricas	Componentes	Versão OO	Versão OA
Acoplamento entre Componentes	Classes	179	142
	Aspectos	-	39
	Total	**179**	**181**
	Diferença	**+1,12%**	
Profundidade da Árvore de Herança	Classes	186	186
	Aspectos	-	2
	Total	**186**	**188**
	Diferença	**+1,08%**	
Falta de Coesão nas Operações	Classes	408	524
	Aspectos	-	0
	Total	**408**	**524**
	Diferença	**+28,43%**	

3.2.3 Métricas de Tamanho

A Tabela 6 exibe os resultados obtidos para as métricas de tamanho no sistema *Java Pet Store*. Os valores obtidos para as métricas Número de Operações e Tamanho do Vocabulário, razoavelmente maiores na versão OA, confirmam as observações feitas em

um estudo anterior [16] que questiona a intuição geral de que aspectos ajudam a produzir programas menores.

A métrica Linhas de Código sofreu um pequeno aumento de aproximadamente 1%. Este resultado tão semelhante entre a versão OO e OA pode ser explicado pelo baixo reuso de código de tratamento de exceções. Outro fator relevante para o LOC não ter diminuído na versão OA foi o *overhead* de implementação inerente ao AspectJ pois, além do código já existente na versão OO a ser executado no tratamento da exceção, é necessário criar conjuntos de pontos de junção para especificar os pontos de junção de interesse e converter as exceções verificadas em não-verificadas. Segue uma ilustração desse *overhead* de implementação, identificado no trecho de código da versão OO do *Java Pet Store* com 13 linhas de código.

```
/* Classe SignOnFilter */
public class SignOnFilter implements Filter {
      . . .
    public void init(FilterConfig config) throws ServletException {
        this.config = config;
        URL protectedResourcesURL = null;
        try {
            protectedResourcesURL = config.getServletContext()
                  .getResource("/WEB-INF/signon-config.xml");
            SignOnDAO dao = new SignOnDAO(protectedResourcesURL);
            signOnErrorPage = dao.getSignOnErrorPage();
            signOnPage = dao.getSignOnPage();
            protectedResources = dao.getProtectedResources();
        } catch (java.net.MalformedURLException ex) {
            System.err.println("SignonFilter: malformed URL exception: "
            + ex);
        }
    }
      . . .
}
```

Em seguida, é exibida a versão OA completa do trecho de código acima. Nota-se que no trecho de código atual, ignorando-se linhas em branco e comentários, existem 21 linhas de código, ou seja, um aumento de aproximadamente 60% com relação à versão OO.

```
/* Versão OA da classe SignOnFilter */
public class SignOnFilter implements Filter {
      . . .
    public void init(FilterConfig config) throws ServletException {
        this.config = config;
        URL protectedResourcesURL = null;
```

```
        protectedResourcesURL = config.getServletContext()
            .getResource("/WEB-INF/signon-config.xml");
        SignOnDAO dao = new SignOnDAO(protectedResourcesURL);
        signOnErrorPage = dao.getSignOnErrorPage();
        signOnPage = dao.getSignOnPage();
        protectedResources = dao.getProtectedResources();
    }
      . . .
}
public aspect SignonWebHandler {
      . . .
      pointcut initHandler() :
            execution(public void SignOnFilter.init(FilterConfig));
      declare soft : MalformedURLException : initHandler();
      void around() : initHandler() {
            try {
                  proceed();
            } catch (MalformedURLException ex) {
                  System.err.println("SignonFilter: malformed URL"+
                        " exception: " + ex);
            }
      }
      . . .
}
```

A métrica Número de Atributos apresentou um pequeno aumento de aproximadamente 1.1%, relativo aos atributos existentes nos aspectos.

Tabela 6 – Métricas de Tamanho para o sistema *Java Pet Store*

Métricas	**Componentes**	**Versão OO**	**Versão OA**
Linhas de Código (LOC)	Classes	17482	15593
	Aspectos	-	2045
	Total	**17482**	**17638**
	Diferença	**+0,89%**	
Número de Atributos	Classes	542	542
	Aspectos	-	6
	Total	**542**	**548**
	Diferença	**+1,11%**	
Número de Operações	Classes	2075	2135
	Aspectos	-	180
	Total	**2075**	**2315**
	Diferença	**+11,57%**	
Tamanho do Vocabulário	Classes	339	339
	Aspectos	-	37
	Total	**339**	**376**
	Diferença	**+10,91%**	

A métrica Número de Operações apresentou um aumento razoável de aproximadamente 11%. Os aspectos foram responsáveis por parte desse aumento devido à criação de comportamentos transversais, mas as classes também contribuíram pois sofreram um aumento aproximado de 3% por causa dos métodos extraídos para expor pontos de junção. Já o aumento de 11% apresentado pela métrica Tamanho do Vocabulário se deve exclusivamente à criação dos aspectos.

A Tabela 7 ilustra os valores coletados para as métricas de tamanho no sistema Telestrada. Todas as quatro métricas obtiveram resultados semelhantes aos obtidos para o sistema *Java Pet Store*, exceto pela métrica Número de Atributos que não sofreu nenhuma alteração durante a refatoração. Porém, esse resultado é totalmente compatível com o obtido para o *Java Pet Store*, já que no Telestrada não houve criação de atributos nos aspectos.

Tabela 7 – Métricas de Tamanho para o sistema Telestrada

Métricas	**Componentes**	**Versão OO**	**Versão OA**
Linhas de Código (LOC)	Classes	3352	2885
	Aspectos	-	449
	Total	**3352**	**3334**
	Diferença	**-0,54%**	
Número de Atributos	Classes	127	127
	Aspectos	-	0
	Total	**127**	**127**
	Diferença	**0%**	
Número de Operações	Classes	423	437
	Aspectos	-	44
	Total	**423**	**481**
	Diferença	**+13,71%**	
Tamanho do Vocabulário	Classes	224	224
	Aspectos	-	18
	Total	**224**	**242**
	Diferença	**+8,04%**	

3.3 Análise dos Resultados

Nesta seção, são ilustradas as análises realizadas a partir dos resultados obtidos com a coleta das métricas.

3.3.1 Reuso

Os resultados obtidos para as versões refatoradas dos sistemas *Java Pet Store* e Telestrada, em comparação com os valores obtidos para as versões originais, indicam que reusar código de tratamento de exceções é uma tarefa difícil. Esses resultados contradizem a intuição geral de que aspectos promovem uma redução no número de LOC do código de tratamento de exceções [26]. Essa observação é reforçada pelo desempenho fraco nas métricas de tamanho das versões refatoradas dos dois sistemas.

Os principais fatores que afetam diretamente o reuso dos tratadores de exceções são: (i) o tipo da exceção a ser tratada; (ii) o código existente no tratador e o seu fluxo de execução, isto é, se o tratador finaliza a execução do método retornando ou propagando a exceção; (iii) a quantidade de informação contextual utilizada; e (iv) o tipo da exceção propagada *versus* as exceções declaradas na cláusula `throws`.

É possível afirmar que o reuso foi muito baixo nos dois sistemas. Um indício disso, por exemplo, é o fato que no PetStore refatorado o valor para LOC aumentou. Com base nos fatores citados acima, é possível compreender os resultados obtidos já que foram encontrados nesses sistemas inúmeros cenários desfavoráveis para o reuso. Os casos mais críticos e que ocorreram com maior freqüência foram: tratadores de exceções contendo informação contextual da classe e tratadores possuindo código '*hardcoded*', como mensagem texto de conteúdo fixo.

Para aumentar o reuso, foi utilizada a estratégia de que cada comportamento transversal no aspecto fosse associado a um número máximo de pontos de junção possível.

Analisando-se as tentativas de reuso utilizadas, é possível concluir que a maneira como a implementação do interesse é quebrada em aspectos (por classe, por pacote, por exceção, etc.) é de extrema importância, pois quanto mais aspectos houver, menores são as oportunidades de reuso. Portanto, como usar um só aspecto pode ser muito ruim para manutenção, desenvolvedores talvez tenham que buscar um meio termo.

Além da dificuldade de obter reuso satisfatório, também se verificou a ocorrência de cenários bastante complexos. Esses cenários demandam análise minuciosa durante a

refatoração para garantir que o comportamento original seja mantido inalterado ao mover o código de tratamento de exceções para aspectos.

3.3.2 Acoplamento

O acoplamento entre componentes mostrou-se praticamente inalterado entre as versões original e refatorada desses sistemas. Porém, ao analisar em mais detalhes, percebeu-se que as classes dos dois sistemas mostraram-se significativamente menos acopladas na versão refatorada. Portanto, analisando-se em conjunto os resultados da métrica Acoplamento entre Componentes e Tamanho do Vocabulário, percebe-se que o uso de aspectos produz um sistema com um número maior de componentes menos fortemente acoplados. De forma mais geral, pode-se afirmar que modularizar tratamento de exceções com aspectos não cria uma quantidade expressiva de acoplamentos novos entre componentes, apenas move acoplamentos já existentes de classes para aspectos. Vale ressaltar que resultados semelhantes a estes podem ser obtidos através do uso de padrões de projeto como *Observer* e *Mediator* [27].

3.3.3 Coesão

Com relação à coesão, verificou-se que na maioria dos casos os componentes com baixa coesão nesses sistemas eram representados pelas classes para as quais foram extraídos métodos para expor pontos de junção. A criação desses métodos pode ser vista como um efeito bastante negativo provocado pela refatoração, pois estes novos métodos foram extraídos apenas como um '*workaround*' para expor alguns pontos de junção para o AspectJ, ou seja, eles não faziam parte do projeto inicial do sistema. Além disso, em grande parte dos casos, esses métodos têm código que, por estar fora do contexto inicial, não faz sentido algum; fato claramente verificado pela dificuldade encontrada ao nomeá-los apropriadamente, ou seja, dificuldade em nomear pode implicar em algo que não deveria ser nomeado [33].

3.4 Considerações Gerais

Um ponto importante a ser citado é que os resultados obtidos para as métricas de separação de interesses, que inclusive foram os melhores dentre todas as métricas, reforçam a afirmação de Lippert & Lopes [26] que o uso de aspectos diminui as interferências entre os interesses no decorrer das linhas de código do programa.

A métrica Número de Operações aumentou nas versões refatoradas desses sistemas. Os fatores responsáveis por esse aumento foram identificados como a criação de novos métodos para expor pontos de junção e a criação dos comportamentos transversais nos aspectos. Contudo, esse acréscimo deve ser analisado com cautela, pois estas operações criadas nos aspectos apresentam código muito mais claro e, principalmente, mais conciso já que não misturam a atividade normal do sistema com o código para tratamento de exceções.

É importante ressaltar que tratamento de exceções não é um interesse "plugável". Baseando-se na definição de "plugabilidade" [34], percebe-se que quando os tratadores de exceções são removidos do código base e criados nos aspectos, obtém-se a separação textual entre os interesses, mas ainda existe dependência entre o código base e os tratadores de exceções, já que a linguagem Java realiza verificação estática das possíveis exceções lançadas. Ou seja, se não for feita a combinação dos aspectos de tratamento de exceções junto com as classes na versão refatorada, o sistema simplesmente não compila.

Enfim, o processo de refatorar o tratamento de exceções exigiu em várias situações soluções complexas e que, provavelmente, nunca seriam usadas na prática. Isso sugere dois problemas: (i) talvez os mecanismos para designação de pontos de junção de interesse de AspectJ não sejam expressivos o suficiente; e (ii) aspectos deveriam ser levados em consideração desde as etapas iniciais no processo de desenvolvimento.

Capítulo 4

TRATAMENTO DE EXCEÇÕES COM POA EM SISTEMAS ORIENTADOS A ASPECTOS

Este capítulo relata o terceiro estudo de caso realizado neste trabalho. Neste estudo, o processo de refatoração é idêntico ao realizado no Capítulo 3, diferindo apenas em alguns detalhes devido ao fato do sistema utilizado já ser originalmente orientado a aspectos. Assim como nos outros dois estudos de caso, métricas são coletadas antes e após a refatoração, sendo então analisadas posteriormente.

Neste estudo, o processo de refatoração também consiste em mover para aspectos todas as ocorrências dos tratadores de exceções existentes nas classes e, neste caso, também nos aspectos originalmente existentes no sistema que implementam outros interesses, como por exemplo, distribuição e persistência. A sistemática utilizada para a refatoração neste estudo de caso teve como maior objetivo a otimização do reuso do código de tratamento de exceções. Ela consistiu na análise prévia de todas as ocorrências desse interesse no código original para identificar padrões de repetição, e em seguida, criação dos aspectos com base nas ocorrências desses padrões.

Este capítulo possui três seções que relatam detalhes do terceiro estudo de caso. Na seção 4.1, informações sobre o sistema *Health Watcher* são sucintamente ilustradas. Na seção 4.2 são expostos os resultados obtidos pelas métricas para ambas as versões original e refatorada desse sistema. Finalmente, na seção 4.3 são discutidas as análises realizadas sobre esses resultados.

4.1 O Sistema *Health Watcher*

O sistema *Health Watcher* é um sistema de informação real que foi desenvolvido para a Secretaria de Saúde da Prefeitura da Cidade do Recife. Suas principais funções são: o

registro e encaminhamento de queixas para o sistema de saúde; e o fornecimento à população de informações sobre unidades de saúde e suas especialidades. Esse sistema disponibiliza uma interface *web* para o registro de reclamações e para consulta de informações de interesse da população que são fornecidas pela Secretaria de Saúde.

Com relação à tecnologia utilizada no seu desenvolvimento, o *Health Watcher* é um sistema orientado a aspectos que foi desenvolvido na linguagem AspectJ. Ele possui aspectos dedicados aos interesses de distribuição, persistência e tratamento de exceções.

A implementação do *Health Watcher* apresenta 6630 LOC e 134 componentes, sendo 98 classes e 36 aspectos.

Assim como nos estudos exibidos no Capítulo 3, no estudo atual a refatoração do sistema também tem o objetivo principal de mover o código destinado a tratamento de exceções das classes para aspectos criados especificamente com este propósito. Além disso, como pode ser visto no exemplo extraído do sistema *Health Watcher* que é ilustrado no Cenário 9 da Seção 5.1.1, também é movido para aspectos o código de tratamento de exceções que aparece nos aspectos não dedicados ao interesse de tratamento de exceções. Neste estudo de caso, a refatoração do sistema *Health Watcher* e a coleta de métricas foram executadas no decorrer do desenvolvimento deste trabalho, seguindo também o mesmo processo utilizado nos outros dois estudos de caso.

O código de tratamento de exceções, originalmente existente nas classes e aspectos, possui aproximadamente 150 blocos *try-catch*, *try-catch-finally* e *try-finally*.

4.2 Resultados

Analogamente aos sistemas *Java Pet Store* e Telestrada descritos no Capítulo 3, após a finalização da etapa de refatoração do sistema *Health Watcher*, aplicou-se o conjunto de métricas para analisar as versões original e refatorada do sistema. Assim como na Seção 3.2, os valores obtidos para as métricas nas duas versões do sistema são apresentados nesta seção através de tabelas. Como o sistema *Health Watcher* é um sistema orientado a aspectos, além de mostrar a contribuição dos aspectos que modularizam o interesse de tratamento de exceções e das classes para o valor final de cada métrica (como no capítulo

anterior), as tabelas desta seção mostram também a contribuição dos aspectos relativos aos interesses de persistência e distribuição originalmente implementados no sistema. Também neste capítulo, o termo **classe** é utilizado para designar tanto classes como interfaces e o termo **componente** designa tanto aspectos quanto classes. O mesmo procedimento de coleta das métricas que foi utilizado no Capítulo 3 também se aplica a esse estudo: métricas de separação de interesses são coletadas manualmente, métrica Número de Atributos é coletada com a ferramenta CASE Together 6.3 [25] e demais métricas são coletadas com a ferramenta AopMetrics [15].

4.2.1 Métricas de Separação de Interesses

A Tabela 8 exibe os resultados obtidos para as métricas de separação de interesses nas duas versões do *Health Watcher*. A versão refatorada apresentou melhorias para as três métricas de separação de interesses. Foi aplicada ao *Health Watcher* a mesma estratégia de refatoração utilizada para os sistemas *Java Pet Store* e Telestrada, na qual todo o código relacionado à implementação do interesse de tratamento de exceções foi movido para aspectos. O diferencial deste estudo de caso, com relação aos anteriores, é o fato de que o sistema original já era orientado a aspectos. Ou seja, já havia aspectos que modularizaram interesses transversais de distribuição e persistência, e também já existiam aspectos que modularizavam parte do código de tratamento de exceções de algumas classes. Portanto, além de mover o código de tratamento de exceções das classes para aspectos dedicados a esse interesse, também foi movido para estes aspectos o código de tratamento de exceções existente nos aspectos cujo interesse transversal não era tratamento de exceções. Nesta refatoração, semelhante ao ocorrido no Telestrada, todo o código relacionado à implementação de tratamento de exceções pôde ser movido para aspectos. Por este motivo, para todas as classes e aspectos de demais interesses da versão refatorada, os valores das métricas de separação de interesses são iguais a zero.

Na métrica Difusão de Interesse por Componentes, o valor coletado para a versão refatorada apresentou uma diminuição de aproximadamente 80% com relação à versão original.

Na métrica Difusão de Interesse por Operações, a versão refatorada também obteve resultados melhores que chegaram a uma diminuição de aproximadamente 50%. Na versão refatorada, os resultados foram iguais ou melhores para todas as classes e aspectos cujo interesse principal não era de tratamento de exceções.

Neste estudo de caso, similarmente ao ocorrido nos estudos anteriores, a métrica Difusão de Interesse por Linhas de Código também foi a que obteve os melhores resultados. A redução na versão refatorada chegou a 100%, ou seja, conseguiu uma separação textual total entre o código que trata exceções e o código responsável pelos outros interesses do sistema.

Tabela 8 – Métricas de Separação de Interesses para o sistema *Health Watcher*

Métricas	Componentes	Versão OA Original	Versão OA Refatorada
Difusão de Interesse por Componentes	Classes	35	0
	Aspectos - demais interesses	7	0
	Aspectos – interesse de tratamento de exceções	5	10
	Total	**47**	**10**
	Diferença	**-78,72%**	
Difusão de Interesse por Operações	Classes	115	0
	Aspectos - demais interesses	12	0
	Aspectos – interesse de tratamento de exceções	9	70
	Total	**136**	**70**
	Diferença	**-48,53%**	
Difusão de Interesse por Linhas de Código	Classes	488	0
	Aspectos - demais interesses	48	0
	Aspectos – interesse de tratamento de exceções	0	0
	Total	**536**	**0**
	Diferença	**-100%**	

4.2.2 Métricas de Acoplamento e Coesão

A Tabela 9 exibe os resultados obtidos para as métricas de acoplamento e coesão nas duas versões do *Health Watcher*. Assim como no *Java Pet Store* e no Telestrada, as versões

original e refatorada desse sistema apresentaram resultados próximos para as métricas de acoplamento.

Nesse sistema, o valor coletado para Acoplamento entre Componentes apresentou uma diminuição aproximada de 1% na versão refatorada. Assim como foi observado nos outros estudos, quanto maior for a quantidade de informação contextual necessária para tratar uma exceção, piores são os valores obtidos para essa métrica.

A métrica Profundidade da Árvore de Herança apresentou-se inalterada na versão refatorada. Esse fato ocorreu porque na versão original do *Health Watcher* já havia uma hierarquia de aspectos dedicados ao interesse de tratamento de exceções que atendia às necessidades da refatoração e, desta forma, não foi necessário criar novos super-aspectos.

A métrica Falta de Coesão nas Operações repetiu neste estudo o mesmo comportamento identificado anteriormente. Ela também foi a métrica que apresentou os piores resultados, pois não apresentou melhoria para nenhum componente. Apesar de manter-se inalterada para a maioria dos componentes, ela apresentou um aumento total de aproximadamente 23%. Foram encontrados no *Health Watcher* os mesmos cenários críticos que foram identificados nos estudos de caso do Capítulo 3: classes para as quais foram extraídos métodos para expor pontos de junção e aspectos contendo atributos.

Tabela 9 – Métricas de Acoplamento e Coesão para o sistema *Health Watcher*

Métricas	**Componentes**	**Versão OA Original**	**Versão OA Refatorada**
Acoplamento entre Componentes	Classes	217	197
	Aspectos - demais interesses	66	61
	Aspectos – interesse de tratamento de exceções	5	27
	Total	**288**	**285**
	Diferença	**-1,04%**	
Profundidade da Árvore de Herança	Classes	69	69
	Aspectos - demais interesses	17	17
	Aspectos – interesse de tratamento de exceções	3	3
	Total	**89**	**89**
	Diferença	**0%**	
Falta de Coesão nas Operações	Classes	766	867
	Aspectos - demais	210	210

	interesses		
	Aspectos – interesse de tratamento de exceções	4	130
	Total	**980**	**1207**
	Diferença	**+23,16%**	

4.2.3 Métricas de Tamanho

A Tabela 10 exibe os resultados obtidos para as métricas de tamanho no sistema *Health Watcher*. Os valores coletados para essas métricas apresentaram resultados próximos para as versões original e refatorada. Porém, a métrica Número de Operações divergiu das demais, apresentando valores razoavelmente maiores na versão refatorada.

Na métrica Linhas de Código, observou-se que o sistema *Health Watcher* apresentou resultados divergentes daqueles encontrados no *Java Pet Store* e no Telestrada, ou seja, a versão refatorada em vez de aumento sofreu uma redução de aproximadamente 6.5%.

A métrica Número de Atributos apresentou um aumento de aproximadamente 2%. Assim como foi identificado nos demais estudos, este acréscimo foi causado exclusivamente pelos atributos criados nos aspectos de tratamento de exceções.

A métrica Número de Operações aumentou na versão refatorada, analogamente ao observado nos estudos de caso anteriores. Neste caso, esse aumento foi de aproximadamente 11.5%. Assim como ocorreu nos demais estudos, os aspectos foram os principais responsáveis devido à criação de comportamentos transversais e as classes também auxiliaram, pois aumentaram em 2% por causa dos métodos extraídos para expor pontos de junção.

Analogamente aos demais estudos, o aumento de aproximadamente 3.5% para a métrica Tamanho do Vocabulário deve ser atribuído exclusivamente à criação dos aspectos.

Tabela 10 – Métricas de Tamanho para o sistema *Health Watcher*

Métricas	Componentes	Versão OA Original	Versão OA Refatorada
Linhas de Código (LOC)	Classes	5732	4641
	Aspectos - demais interesses	812	701
	Aspectos - interesse de tratamento de exceções	86	853

	Total	**6630**	**6195**
	Diferença	**-6,56%**	
Número de Atributos	Classes	152	152
	Aspectos - demais interesses	12	12
	Aspectos – interesse de tratamento de exceções	3	7
	Total	**167**	**171**
	Diferença	**+2,4%**	
Número de Operações	Classes	542	553
	Aspectos - demais interesses	104	104
	Aspectos - interesse de tratamento de exceções	9	73
	Total	**655**	**730**
	Diferença	**+11,45%**	
Tamanho do Vocabulário	Classes	98	98
	Aspectos - demais interesses	31	31
	Aspectos - interesse de tratamento de exceções	5	10
	Total	**134**	**139**
	Diferença	**+3,73%**	

4.3 Análise dos Resultados

Nesse terceiro estudo de caso, como o sistema utilizado já possuía aspectos que modularizavam outros interesses diferentes de tratamento de exceções, foi mais difícil garantir a integridade do comportamento original na versão refatorada do sistema. Esse evento ocorreu porque, em sistemas onde vários interesses distintos são “aspectizados”, existe uma grande possibilidade de que um mesmo ponto de junção seja capturado por vários conjuntos de pontos de junção pertencentes a aspectos de interesses distintos. Assim, é necessário definir a ordem de execução dos múltiplos aspectos nesses pontos de junção compartilhados, para garantir a integridade do comportamento original do sistema.

No sistema original utilizado nesse terceiro estudo de caso, já havia aspectos dedicados ao interesse de tratamento de exceções que modularizavam parte do código de tratamento de exceções existente nas classes. Portanto, foi necessário analisar meticulosamente a estrutura

existente, adaptando-a quando necessário, para atingir o objetivo de mover completamente o código relacionado à implementação do interesse de tratamento de exceções para aspectos dedicados a esse interesse.

Conforme foi descrito na seção 3.3.4, tratamento de exceções não é um interesse "plugável". Ao se "aspectizar" o código de tratamento de exceções existente em um aspecto, aumentam-se as chances do sistema não compilar. Esse fato ocorre porque, caso esse aspecto refatorado não seja combinado corretamente com o aspecto de tratamento de exceções, além desses dois componentes não compilarem, também não irão compilar as classes "aspectizadas" pelo aspecto refatorado, ou seja, esse problema pode se propagar por todo o sistema.

Diferente do que foi observado na Seção 3.3, o reuso do código de tratamento de exceções foi menos difícil no sistema desse terceiro estudo de caso. Esse fato pode se observado através da métrica Linhas de Código, que em vez de aumentar, sofreu uma redução.

Como foi explicado no capítulo anterior, o reuso pode ser influenciado pela política adotada para a criação de aspectos. Desta forma, com o objetivo de tentar melhorar o reuso do código de tratamento de exceções, foram inicialmente analisadas todas as ocorrências desse interesse no sistema para identificar padrões de repetição de código, e em seguida, os aspectos foram criados com base nas ocorrências desses padrões. Durante essa análise, buscou-se também identificar oportunidades de utilizar a herança simples entre aspectos, criando-se o máximo possível de super-aspectos.

Finalmente, analisando-se o código desses sistemas é possível supor que a principal causa da menor dificuldade em se obter reuso nesse terceiro sistema esteja relacionada com o fato dos tratadores de exceções serem comparativamente mais simples no *Health Watcher* do que nos outros dois sistemas. Esse menor grau de complexidade nos tratadores de exceções pode ser explicado pelo fato de que o sistema *Health Watcher*, dentre os três sistemas utilizados neste trabalho, é o que possui código de tratamento de exceções melhor estruturado.

Capítulo 5

CENÁRIOS DE TRATAMENTO DE EXCEÇÕES COM POA

Este capítulo apresenta uma coletânea de exemplos de cenários que foram identificados durante a realização dos estudos de caso descritos nos Capítulos 3 e 4. Esses cenários representam situações recorrentes com as quais um desenvolvedor incumbido de refatorar para aspectos o código de tratamento de exceções de um sistema se defrontará. O principal objetivo deste capítulo é auxiliar desenvolvedores nesta tarefa, tanto mostrando situações em que o uso de aspectos melhora a qualidade do código do sistema quanto evidenciando casos em que modularizar tratamento de exceções com aspectos não é uma solução adequada.

Os cenários apresentados neste capítulo foram classificados em duas categorias: **cenários de refatoração** (Seção 5.1), que estão relacionados ao próprio processo de refatoração; e **cenários de reuso** (Seção 5.2), que destacam a (im)possibilidade de reuso de tratadores dentro de um sistema, durante o processo de refatoração.

5.1 Cenários de Refatoração

Com o intuito de facilitar a compreensão dos diferentes cenários identificados, estes foram classificados em **cenários benéficos** e **cenários prejudiciais**. Essa classificação foi definida com base na qualidade do código final obtido com a "aspectização". Desta forma, cenários benéficos representam situações em que código de tratamento de exceções pode ser refatorado para aspectos de forma elegante, produzindo código final de qualidade superior. Essa qualidade se reflete na influência positiva desses cenários nos valores obtidos para as métricas empregadas no estudo. Por outro lado, cenários prejudiciais representam situações para as quais a modularização para aspectos é prejudicial, isto é,

insere mais danos no código do que benefícios. Mover código de tratamento de exceções para aspectos nos cenários prejudiciais implica em piorar os valores das métricas. Em geral, as métricas mais afetadas por esses cenários são **Falta de Coesão nas Operações, Linhas de Código, Número de Atributos** e **Número de Operações**.

Os cenários apresentados nesta seção seguem um mesmo formato. Primeiro, é fornecida uma descrição informal sobre o cenário e suas diferenças com relação a outros cenários que foram apresentados antes. Em seguida, é apresentado um exemplo concreto do cenário, na forma de código extraído de algum dos sistemas-alvo dos estudos de caso. Esses trechos de código incluem tanto o código do sistema original quanto o do sistema refatorado.

5.1.1 Cenários Benéficos

Cenário 1

Este é o cenário mais simples identificado nos estudos de caso. Como o corpo inteiro do método é interno a um bloco `try-catch`, é fácil mover o código do tratador para um comportamento transversal. Além disso, já que o bloco `catch` termina sua execução lançando uma exceção, o tratador na versão refatorada pode ser implementado como um comportamento transversal do tipo *after throwing*, mais simples do que um comportamento transversal do tipo *around*. Se o bloco *catch* não lançasse exceções, seria necessário utilizar um comportamento transversal do tipo *around*. Comportamentos transversais do tipo *after throwing* não permitem que o fluxo de controle de um sistema seja modificado, ou seja, se uma exceção for lançada a partir de um ponto de junção selecionado, após a execução de um comportamento transversal do tipo *after throwing*, a propagação dessa exceção continua, mesmo que o comportamento transversal não lance exceções. Outro fator que torna esse cenário simples é o fato de a exceção encontrada pelo tratador, `XMLDocumentException`, não ter que ser convertida em não-verificada por já ser declarada explicitamente na interface do método.

O trecho de código apresentado a seguir, extraído do sistema *Java Pet Store*, ilustra esse cenário.

Código Original

```
public class OrderApproval  {
      . . .
      public static OrderApproval fromXML(String buffer,
```

```
      URL entityCatalogURL, boolean validating) throws
      XMLDocumentException {
      try {
            return fromXML(new StreamSource(new StringReader(buffer)),
            entityCatalogURL, validating);
    } catch (XMLDocumentException exception) {
      System.err.println(exception.getRootCause().getMessage());
      throw new XMLDocumentException(exception);
    }
  }
      . . .
}
```

Código Refatorado

```
public class OrderApproval  {
      . . .
      public static OrderApproval fromXML(String buffer, URL
            entityCatalogURL, boolean validating) throws
      XMLDocumentException {
            return fromXML(new StreamSource(new StringReader(buffer)),
            entityCatalogURL, validating);
      }
      . . .
}
/* Aspecto */
public aspect XmlDocumentsHandler {
      . . .
      public pointcut afterWithPrintXMLDocumentExceptionHandler() :
            execution(* OrderApproval OrderApproval.fromXML(..));

      after() throwing(XMLDocumentException exception) throws
      XMLDocumentException :
          afterWithPrintXMLDocumentExceptionHandler() {
        System.err.println(exception.getRootCause().getMessage());
        throw new XMLDocumentException(exception);
      }
      . . .
}
```

Cenário 2

Este também é um cenário bastante simples. Da mesma forma que no Cenário 1, o tratador de exceções também é modularizado com um comportamento transversal do tipo *after throwing*. Neste caso, porém, as exceções verificadas `CreateException` e `NamingException` precisam ser convertidas em não-verificadas, já que são lançadas pelo corpo do método mas não aparecem em sua cláusula `throws`. O trecho de código apresentado a seguir, extraído do sistema *Java Pet Store*, ilustra esse cenário.

Código Original

```
public class CreateUserServlet extends HttpServlet {
      . . .
     private SignOnLocal getSignOnEjb() throws ServletException {
         SignOnLocal signOn = null;
         try {
            InitialContext ic = new InitialContext();
            Object o = ic.lookup("java:comp/env/ejb/SignOn");
            SignOnLocalHome home =(SignOnLocalHome)o;
            signOn = home.create();
         } catch (javax.ejb.CreateException cx) {
            throw new ServletException("Failed to Create SignOn EJB:
            caught " + cx);
         } catch (javax.naming.NamingException nx) {
            throw new ServletException("Failed to Create SignOn EJB:
            caught " + nx);
        }
        return signOn;
     }
      . . .
}
```

Código Refatorado

```
public class CreateUserServlet extends HttpServlet {
      . . .
     private SignOnLocal getSignOnEjb() throws ServletException {
         SignOnLocal signOn = null;
         InitialContext ic = new InitialContext();
         Object o = ic.lookup("java:comp/env/ejb/SignOn");
         SignOnLocalHome home =(SignOnLocalHome)o;
         signOn = home.create();
         return signOn;
     }
      . . .
}
/* Aspecto */
public aspect SignonWebHandler {
      . . .
      pointcut getSignOnEjbHandler() :
         execution(private SignOnLocal CreateUserServlet.getSignOnEjb());
      declare soft : CreateException : getSignOnEjbHandler();
      declare soft : NamingException : getSignOnEjbHandler();

      after() throwing(Exception ex) throws ServletException :
            getSignOnEjbHandler() {
      if (ex instanceof CreateException ||
            ex instanceof NamingException)
        throw new ServletException(
            "Failed to Create SignOn EJB: caught " + ex);
      }
      . . .
}
```

Cenário 3

Neste cenário, o tratador de exceções precisa ser "aspectizado" utilizando-se um comportamento transversal do tipo *around*, pois a execução do tratador não termina lançando uma exceção. Além disso, similarmente ao exemplo anterior, as exceções verificadas lançadas pelo corpo do método que não aparecem em sua cláusula `throws` precisam ser convertidas em não-verificadas. O trecho de código apresentado a seguir, extraído do sistema Telestrada, ilustra esse cenário.

Código Original

```
public class ConnectionPool extends GenericPool {
      . . .
  public void log(String aMsg) {
    Object[] params = new Object[1];
    params[0] = aMsg;
    try {
      logMethod.invoke(logObject, params);
    } catch (Exception e) {
    } // ignore exceptions when logging
  }
      . . .
}
```

Código Refatorado

```
public class ConnectionPool extends GenericPool {
      . . .
  public void log(String aMsg) {
    Object[] params = new Object[1];
    params[0] = aMsg;
    logMethod.invoke(logObject, params);
  }
      . . .
}
/* Aspecto */
privileged public aspect ConnectionPoolHandlers {
      . . .
  pointcut logHandler() : execution(public void log(..));
  declare soft : InvocationTargetException : logHandler();
  declare soft : IllegalAccessException : logHandler();
  void around() : logHandler() {
    try {
      proceed();
    } catch (Exception e) {
    } // ignore exceptions when logging
  }
      . . .
}
```

Cenário 4

Este é um cenário simples que aparenta ser mais complexo devido à existência de um bloco `try-catch` dentro de um bloco `finally`. Como o objetivo do processo de refatoração neste trabalho é separar o código responsável pela atividade normal do código da atividade excepcional e os finalizadores de um programa são parte da sua atividade excepcional, o corpo do bloco `finally` pode ser movido para um comportamento transversal da mesma maneira como é feito no cenário anterior.

O trecho de código apresentado a seguir, extraído do sistema *Health Watcher*, ilustra esse cenário.

Código Original

```
public class ConcurrencyManager {
      . . .
    public synchronized void endExecution(Object key) {
        try {
            . . .
        } catch(Exception ex) {
            System.out.println(ERROR_MESSAGE+ex.getMessage());
        } finally {
            try {
                notifyAll();
            } catch(Exception e) {
              throw new RuntimeException(ERROR_MESSAGE+e.getMessage());
            }
        }
    }
      . . .
}
```

Código Refatorado

```
public class ConcurrencyManager {
      . . .
    public synchronized void endExecution(Object key) {
            . . .
    }
      . . .
}
/* Aspecto */
privileged aspect ConcurrencyControlExceptionHandling  {
      . . .
    pointcut endExecutionHandler() :
        execution(* ConcurrencyManager.endExecution(..));

    void around(ConcurrencyManager cm) :
       endExecutionHandler() && target(cm) {
        try {
            proceed(cm);
        } catch(Exception ex) {
            System.out.println(ConcurrencyManager.ERROR_MESSAGE+
```

```
            ex.getMessage());
        } finally {
            try {
                cm.notifyAll();
            } catch(Exception e) {
                throw new RuntimeException(
                  ConcurrencyManager.ERROR_MESSAGE+e.getMessage());
            }
        }
    }
      . . .
}
```

Cenário 5

Neste cenário, apesar do bloco `try-catch` não envolver todo o corpo do método, é vantajoso mover o tratador de exceções para um aspecto. Como o tratador termina sua execução propagando uma exceção, é possível considerar que o bloco `try-catch` engloba todo o corpo do método, já que o término da execução do tratador implica no término da execução do método. Isso torna possível reduzir este cenário para os Cenários 1 ou 2, dependendo da necessidade de se converter a exceção em não-verificada.

O trecho de código apresentado a seguir, extraído do sistema Telestrada, ilustra esse cenário.

Código Original

```
public class TrechoConsulta implements ITrechoObj {
      . . .
public ArrayList execute(TrechoDAO dao, GenericOperations aGenOp,
long idUfRodovia) throws ITrechoConsultaErroException,
ITrechosNaoExistentesException {
            System.out.println("TrechoConsulta: sendo executado");
            ArrayList list = null;
            try {
                  System.out.println("TrechoConsulta: Selecionando lista
                  de rodovias");
                  list = dao.selectAllTrecho(aGenOp, idUfRodovia);
            } catch (SQLException e) {
                  ITrechoConsultaErroException ex = excep.handle(e);
                  throw ex;
            }
            if (list == null || list.isEmpty()) {
                  ITrechosNaoExistentesException ex =
                  new TrechosNaoExistentesException();
                  throw ex;
            }
            return list;
      }
      . . .
}
```

Código Refatorado

```
public class TrechoConsulta implements ITrechoObj {
      . . .
public ArrayList execute(TrechoDAO dao, GenericOperations aGenOp, long
idUfRodovia) throws ITrechoConsultaErroException,
ITrechosNaoExistentesException {
            System.out.println("TrechoConsulta: sendo executado");
            ArrayList list = null;
System.out.println("TrechoConsulta: Selecionando lista de
rodovias");
            list = dao.selectAllTrecho(aGenOp, idUfRodovia);
            if (list == null || list.isEmpty()) {
                  ITrechosNaoExistentesException ex =
new TrechosNaoExistentesException();
                  throw ex;
            }
            return list;
      }
      . . .
}
/* Aspecto */
public aspect TrechoConsultaHandler {
      . . .
  pointcut executeHandler() :
    execution(public ArrayList TrechoConsulta.execute(..));
  declare soft : SQLException : executeHandler();
  after() throwing (SQLException e) throws ITrechoConsultaErroException :
        executeHandler() {
     ITrechoConsultaErroException ex = new TrechoConsultaErroException();
     throw ex;
  }
      . . .
}
```

Cenário 6

Esse cenário é semelhante ao Cenário 4, com a diferença de que os blocos `try-catch`, `try-finally` ou `try-catch-finally` aparecem aninhados dentro do código de atividade normal e, portanto, precisam ser "aspectizados". Neste caso, os blocos `try-catch` aninhados podem ser associados ao mesmo conjunto de pontos de junção. Porém, é importante que os comportamentos transversais correspondentes a esses blocos apareçam no texto do código do aspecto seguindo a ordem correta, isto é, o comportamento transversal do bloco mais interno deve aparecer textualmente antes do comportamento transversal relativo ao bloco mais externo.

O trecho de código apresentado a seguir, extraído do sistema *Health Watcher*, ilustra esse cenário.

Código Original

```
public class ComplaintRepositoryRDBMS implements IComplaintRepository {
      . . .
public void update(Complaint complaint) throws ObjectNotFoundException,
ObjectNotValidException {
        try {
            if (complaint != null) {
                try {
. . .
                } catch (SQLException e) {
                    throw new PersistenceSoftException(e);
                }
            } else {
throw new ObjectNotValidException(ExceptionMessages.EXC_NULO);
            }
        } catch (PersistenceMechanismException e) {
            throw new PersistenceSoftException(e);
        }
    }
      . . .
}
```

Código Refatorado

```
public class ComplaintRepositoryRDBMS implements IComplaintRepository {
      . . .
public void update(Complaint complaint) throws ObjectNotFoundException,
ObjectNotValidException {
        if (complaint != null) {
            . . .
        } else {
throw new ObjectNotValidException(ExceptionMessages.EXC_NULO);
        }
    }
      . . .
}
/* Aspecto */
privileged aspect DataCollectionsExceptionHandling {
      . . .
    pointcut updateComplaintHandler() :
      execution(* ComplaintRepositoryRDBMS.update(Complaint));
    declare soft : SQLException : updateComplaintHandler();
    declare soft : PersistenceMechanismException :
updateComplaintHandler();
    after() throwing(SQLException e) : updateComplaintHandler() {
        throw new PersistenceSoftException(e);
    }
after() throwing(PersistenceMechanismException e) :
updateComplaintHandler() {
        throw new PersistenceSoftException(e);
    }
      . . .
}
```

Cenário 7

Neste cenário, assim como no Cenário 5, o bloco `try-catch` não envolve todo o corpo do método. A diferença entre os dois cenários é que neste, a execução do tratador termina com um comando `return`. Como, da mesma forma que no Cenário 5, o término da execução do tratador implica no término da execução do método, os dois cenários são similares. A única diferença significativa está no fato de, neste cenário, o comportamento transversal para o qual o código de tratamento de exceções é movido ter que ser do tipo *around*, já que o tratador não termina com o lançamento de uma exceção.

O trecho de código a seguir, extraído do sistema *Java Pet Store*, ilustra esse cenário.

Código Original

```
public class PieChartPanel extends JPanel implements
PropertyChangeListener {
      . . .
    private void updateModelDates() {
        Date startDate = null;
        Date endDate = null;
        try {
            . . .
        } catch (ParseException e) {
            JOptionPane.showMessageDialog(this, PetStoreAdminClient.
            getString("DateFormatErrorDialog.message"),
            PetStoreAdminClient.getString("DateFormatErrorDialog.title"),
            JOptionPane.ERROR_MESSAGE);
            return;
        }
        pieChartModel.setDates(startDate, endDate);
    }
      . . .
}
```

Código Refatorado

```
public class PieChartPanel extends JPanel implements
PropertyChangeListener {
      . . .
    private void updateModelDates() {
        Date startDate = null;
        Date endDate = null;
            . . .
        pieChartModel.setDates(startDate, endDate);
    }
      . . .
}
/* Aspecto */
public aspect AdminClientHandler {
      . . .
      pointcut updateModelDatesHandler() :
```

```
       execution(private void PieChartPanel.updateModelDates());

      declare soft : ParseException : updateModelDatesHandler();

      void around(PieChartPanel pcp) : updateModelDatesHandler() &&
      target(pcp) {
       try {
          proceed(pcp);
       } catch (ParseException e) {
            JOptionPane.showMessageDialog(pcp, PetStoreAdminClient.
            getString("DateFormatErrorDialog.message"),
            PetStoreAdminClient.getString("DateFormatErrorDialog.title"),
            JOptionPane.ERROR_MESSAGE);
       }
    }
      . . .
}
```

Cenário 8

Neste cenário, há um bloco `try-catch` que não engloba todo o corpo do método no qual aparece. Além disso, a execução do tratador não termina com o lançamento de uma exceção ou com um comando `return`. Um cenário como esse pode ser benéfico ou prejudicial, dependendo da facilidade com a qual se consegue selecionar o código dentro do bloco `try` com conjuntos de pontos de junção. Neste cenário, em especial, é possível definir um conjunto de pontos de junção que captura o ponto de junção relativo ao código existente no bloco `try`, pois ele é composto de apenas 1 comando.

O trecho de código apresentado a seguir, extraído do sistema *Java Pet Store*, ilustra esse cenário.

Código Original

```
public class TemplateServlet extends HttpServlet {
      . . .
    private void initScreens(ServletContext context, String language) {
        URL screenDefinitionURL = null;
        try {
            screenDefinitionURL = context.getResource(
"/WEB-INF/screendefinitions_" + language + ".xml");
        } catch (java.net.MalformedURLException ex) {
System.err.println("TemplateServlet:malformed URL exception:" + ex);
        }
            . . .
    }
      . . .
}
```

Código Refatorado

```
public class TemplateServlet extends HttpServlet {
      . . .
    private void initScreens(ServletContext context, String language) {
        URL screenDefinitionURL = null;
        screenDefinitionURL = context.getResource(
"/WEB-INF/screendefinitions_" + language + ".xml");
. . .
    }
      . . .
}
/* Aspecto */
public aspect WafViewTemplateHandler {
      . . .
   pointcut initScreensGetResourceHandler() :
      call(URL ServletContext.getResource(String)) &&
      withincode(* TemplateServlet.initScreens(..));
      declare soft : MalformedURLException :
           initScreensGetResourceHandler();
      URL around() : initScreensGetResourceHandler() {
            try {
                  return proceed();
            } catch (MalformedURLException ex) {
                System.err.println("TemplateServlet: malformed URL "+
                "exception: " + ex);
                return null;
            }
      }
      . . .
}
```

Cenário 9

Neste cenário, há um bloco `try-catch` em um aspecto cujo interesse não é de tratamento de exceções. Assim, da mesma forma que foi feito com as classes, o código de tratamento de exceções também é movido para um aspecto dedicado a tratamento de exceções. Este cenário é similar ao Cenário 2, diferenciando-se apenas pelo fato do componente "aspectizado" ser um aspecto e não uma classe.

O trecho de código apresentado a seguir, extraído do sistema *Health Watcher*, ilustra esse cenário.

Código Original

```
/* Aspecto de Distribuição - Original */
privileged public aspect ClientDistributedParametrizedDataLoading {
      . . .
    IteratorHW around() throws ObjectNotFoundException:
       flatGetHealthUnitListCall()
    {
        try {
          IRemoteFacade healthWatcher = (IRemoteFacade)
```

```
          HealthWatcherClientSideAspect.aspectOf().getRemoteFacade();
            return healthWatcher.flagGetHealthUnitList();
        } catch(RemoteException ex) {
            throw new SoftException(ex);
        }
    }
      . . .
}
```

Código Refatorado

```
/* Aspecto de Distribuição - Refatorado */
privileged public aspect ClientDistributedParametrizedDataLoading {
      . . .
    IteratorHW around() throws ObjectNotFoundException:
        flatGetHealthUnitListCall()
    {
        IRemoteFacade healthWatcher = (IRemoteFacade)
          HealthWatcherClientSideAspect.aspectOf().getRemoteFacade();
        return healthWatcher.flagGetHealthUnitList();
    }
      . . .
}
/* Aspecto de Tratamento de Exceções */
privileged aspect OtherAspectsExceptionHandlingAspect {
      . . .
    pointcut clientDistributedParametrizedDataLoadingHandler() :
        adviceexecution() &&
        within(ClientDistributedParametrizedDataLoading);
    declare soft : RemoteException :
        within(ClientDistributedParametrizedDataLoading);
    after() throwing(RemoteException ex) :
       clientDistributedParametrizedDataLoadingHandler() {
          throw new SoftException(ex);
    }
      . . .
}
```

5.1.2 Cenários Prejudiciais

Cenário 10

Assim como no Cenário 8, neste cenário há um bloco `try-catch` que não engloba todo o corpo do método no qual aparece e a execução do tratador não termina com o lançamento de uma exceção ou com um comando `return`. Neste caso, a solução encontrada foi extrair um método contendo a lógica do bloco `try-catch` para expor um ponto de junção que AspectJ consegue capturar.

O trecho de código apresentado a seguir, extraído do sistema *Java Pet Store*, apresenta esse cenário.

Código Original

```
public class ClientStateFlowHandler implements FlowHandler {
      . . .
    public String processFlow(HttpServletRequest request)
        throws FlowHandlerException {
            . . .
            while (it.hasNext()) {
                  . . .
                  try {
                        . . .
                    } catch (java.io.OptionalDataException ode) {
                        System.err.println("ClientCacheLinkFlowHandler "+
                        "caught: " + ode);
                    } catch (java.lang.ClassNotFoundException cnfe) {
                        System.err.println("ClientCacheLinkFlowHandler "+
                        "caught: " + cnfe);
                    } catch (java.io.IOException iox) {
                        System.err.println("ClientCacheLinkFlowHandler "+
                        "caught: " + iox);
                    }
                  . . .
            }
            . . .
        }
        return forwardScreen;
    }
      . . .
}
```

Código Refatorado

```
public class ClientStateFlowHandler implements FlowHandler {
      . . .
    public String processFlow(HttpServletRequest request)
        throws FlowHandlerException {
        String forwardScreen = request.getParameter("referring_screen");
            . . .
            while (it.hasNext()) {
                  . . .
                  internalProcessFlow(...);
                  . . .
            }
            . . .
        }
        return forwardScreen;
    }
    private void internalProcessFlow(...) {
            . . .
    }
      . . .
}
```

```
/* Aspecto */
public aspect WafControllerWebHandler {
      . . .
    pointcut internalProcessFlowHandler() :
       execution(* ClientStateFlowHandler.internalProcessFlow(..));
    declare soft : IOException : internalProcessFlowHandler();
      . . .
    void around() : internalProcessFlowHandler() {
      try {
            proceed();
        } catch (OptionalDataException ode) {
            System.err.println("ClientCacheLinkFlowHandler caught: " +
            ode);
        } catch (ClassNotFoundException cnfe) {
            System.err.println("ClientCacheLinkFlowHandler caught: " +
            cnfe);
        } catch (IOException iox) {
            System.err.println("ClientCacheLinkFlowHandler caught: " +
            iox);
        }
    }
      . . .
}
```

Cenário 11

Neste cenário, o finalizador de um bloco `try-catch-finally` acessa variáveis locais do método. Dentre algumas soluções possíveis, decidiu-se por uma que utiliza um comportamento transversal do tipo *after returning* para armazenar a referência da variável local do método em uma variável do aspecto. Desta forma, é possível utilizar essa variável no comportamento transversal que modulariza o tratator de exceções, já que se torna um atributo do aspecto. Para garantir consistência no caso de múltiplas *threads* de execução, o valor da variável é armazenado em uma tabela indexada pelo objeto que representa a *thread* sendo executada. Outra complicação é o fato de dois comportamentos transversais diferentes, um responsável por capturar o valor da variável local e outro relativo ao finalizador, estarem associados ao mesmo conjunto de pontos de junção. Para garantir a ordem correta de execução dos comportamentos transversais, o comportamento transversal a ser executado primeiro (o que captura o valor da variável) deve aparecer antes, lexicamente, no código do aspecto.

Apesar de tornar possível mover o código relativo ao comportamento excepcional do método `main()` para um aspecto, a solução adotada apresenta diversas desvantagens: (i) exige uma tabela adicional para cada variável local cujo valor precisa ser salvo; (ii) não

funciona se a variável é atualizada diversas vezes, de maneira que não podem ser descritas com conjuntos de pontos de junção; e (iii) cria um aspecto que é fortemente acoplado com o código OO. Uma outra solução possível é extrair o corpo do bloco `try` para um novo método parametrizado com a variável local que o aspecto precisa acessar. Essa solução apresenta o problema (iii), mencionado acima, além de exigir que seja criado um método que não faz sentido por si só e que não é parte do projeto original do sistema.

O trecho de código apresentado a seguir, extraído do sistema *Health Watcher*, ilustra esse cenário.

Código Original

```
public class GerenciaTabelas {
      . . .
    public static void main(String[] args) {
        PersistenceMechanismRDBMS pm = null;
        try {
            pm = PersistenceMechanismRDBMS.getInstance();
            . . .
        }
        catch (Exception ex) {
            . . .
        } finally {
                . . .
            try {
                pm.releaseCommunicationChannel();
            } catch (Exception ex) {
                  . . .
            }
                  . . .
        }
    }
      . . .
}
```

Código Refatorado

```
public class GerenciaTabelas {
      . . .
    public static void main(String[] args) {
        PersistenceMechanismRDBMS pm = null;
        pm = PersistenceMechanismRDBMS.getInstance();
            . . .
    }
      . . .
}
/* Aspecto */
privileged aspect PersistentUtilExceptionHandling {
      . . .
    Map persistenceMechanismRDBMS = new HashMap();
    pointcut persistenceMechanismRDBMSGetInstanceHandler() :
       call(* PersistenceMechanismRDBMS.getInstance()) &&
       withincode(* GerenciaTabelas.main(String[]));
```

```
    pointcut gerenciaTabelasMainHandler() :
       execution(* GerenciaTabelas.main(String[]));
    declare soft : Exception : gerenciaTabelasMainHandler();
    after() returning(PersistenceMechanismRDBMS pm) :
    persistenceMechanismRDBMSGetInstanceHandler() {
       //Save inner method variable to local(multi-thread)
       persistenceMechanismRDBMS.put(String.valueOf(
       Thread.currentThread().getId()), pm);
    }
    void around() : gerenciaTabelasMainHandler() {
       try {
           proceed();
       } catch (Exception ex) {
            . . .
       } finally {
          PersistenceMechanismRDBMS pm = (PersistenceMechanismRDBMS)
          persistenceMechanismRDBMS.get(String.valueOf(
          Thread.currentThread().getId()));
                  . . .
          try {
              pm.releaseCommunicationChannel();
          } catch (Exception ex) {
                  . . .
          }
                  . . .
    }
      . . .
}
```

Cenário 12

Este cenário é semelhante ao Cenário 6, isto é, os blocos `try-catch`, `try-finally` ou `try-catch-finally` aparecem aninhados dentro no corpo de um método. A diferença é que neste cenário existe código adicional antes e/ou após os blocos aninhados. Neste caso, é necessário extrair um novo método para cada um desses blocos, a fim de expor pontos de junção que possam ser capturados por AspectJ. Segue um exemplo deste cenário encontrado no sistema *Health Watcher*.

Código Original

```
public class HealthUnitRepositoryRDBMS implements IHealthUnitRepository {
      . . .
    public HealthUnit search(int code) throws ObjectNotFoundException {
        HealthUnit hu = null;
        try {
                  . . .
              try {
                  . . .
              } catch (ObjectNotFoundException ex) { }
           }
                  . . .
        } catch (PersistenceMechanismException e) {
            throw new PersistenceSoftException(e);
```

```
        } catch (SQLException e) {
            throw new PersistenceSoftException(e);
        } finally {
            . . .
        }
        return hu;
    }
      . . .
}
```

Código Refatorado

```
public class HealthUnitRepositoryRDBMS implements IHealthUnitRepository {
      . . .
    public HealthUnit search(int code) throws ObjectNotFoundException {
        HealthUnit hu = null;
      . . .
        internalGenerateSpecialtyRepositoryArray(...);
            . . .
        return hu;
    }
    private void internalGenerateSpecialtyRepositoryArray(...) {
      . . .
    }
      . . .
}
/* Aspecto */
privileged aspect DataCollectionsExceptionHandling {
      . . .
    pointcut searchHealthUnitHandler() :
      execution(* HealthUnitRepositoryRDBMS.search(..));
    pointcut internalGenerateSpecialtyRepositoryArrayHandler() :
      execution(* HealthUnitRepositoryRDBMS
.internalGenerateSpecialtyRepositoryArray(..));
    declare soft : SQLException : searchHealthUnitHandler();
    declare soft : PersistenceMechanismException :
searchHealthUnitHandler();
    declare soft : ObjectNotFoundException :
internalGenerateSpecialtyRepositoryArrayHandler();
    void around() : internalGenerateSpecialtyRepositoryArrayHandler() {
      try {
            proceed();
      } catch(ObjectNotFoundException ex) { }
    }
    Object around(Object obj) : searchHealthUnitHandler() && target(obj){
      try {
            return proceed(obj);
      } catch (PersistenceMechanismException e) {
            throw new PersistenceSoftException(e);
      } catch (SQLException e) {
            throw new PersistenceSoftException(e);
      } finally {
            . . .
      }
    }
      . . .
}
```

5.2 Cenários de Reuso

Esses cenários foram classificados da seguinte forma: **cenários de sucesso** e **cenários de insucesso**. Essa classificação baseia-se na viabilidade ou inviabilidade de se reutilizar os tratadores de exceções. Assim, cenários de sucesso são trechos de código de tratamento de exceções que se repetem em vários componentes e que podem ser reutilizados no código "aspectizado", sendo mapeados para um único tratador. Ou seja, um único comportamento transversal é capaz de modularizar diversos blocos `try-catch`. Por outro lado, cenários de insucesso representam situações para as quais esse reuso não é possível porque, apesar da semelhança entre os tratadores, pequenos detalhes fazem com que cada tratador tenha que ser "aspectizado" separadamente. Ou seja, um bloco `try-catch` demanda a criação de um comportamento transversal totalmente dedicado a si mesmo.

Os cenários nesta seção seguem um mesmo formato. Primeiro, é fornecida uma descrição informal sobre o cenário, salientando suas diferenças com relação a outros cenários que tenham sido apresentados antes. Depois, é apresentado um exemplo concreto do cenário, na forma de código extraído de algum dos sistemas utilizados nos estudos de caso. Esses trechos de código incluem o código do sistema original e também o código do sistema refatorado.

5.2.1 Cenários de Sucesso

Cenário 1

Este é o cenário mais apropriado para reuso identificado nos estudos de caso. Neste caso, existem várias ocorrências do mesmo bloco `catch` em diversos métodos de uma mesma classe, `CatalogEJB`, sendo possível mover o tratador para um único comportamento transversal. Como todos os pontos de junção são relacionados a métodos pertencentes a uma classe apenas, também é possível definir apenas um conjunto de pontos de junção no aspecto. Além disso, este também é um cenário de refatoração muito simples, que nem sequer precisa converter a exceção encontrada pelo tratador, `CatalogDAOSysException`, em não-verificada por já ser uma exceção desse tipo.

O trecho de código apresentado a seguir, extraído do sistema *Java Pet Store*, ilustra este cenário.

Código Original

```
public class CatalogEJB implements SessionBean {
      . . .
    public Category getCategory(String categoryID, Locale l) {
        try {
            return dao.getCategory(categoryID, l);
        }
        catch (CatalogDAOSysException se) {
            throw new EJBException(se.getMessage());
        }
    }
      . . .
    public Page searchItems(String searchQuery, int start,
                            int count, Locale l) {
        try {
            return dao.searchItems(searchQuery, start, count, l);
        }
        catch (CatalogDAOSysException se) {
            throw new EJBException(se.getMessage());
        }
    }
      . . .
}
```

Código Refatorado

```
public class CatalogEJB implements SessionBean {
      . . .
    public Category getCategory(String categoryID, Locale l) {
        return dao.getCategory(categoryID, l);
    }
      . . .
    public Page searchItems(String searchQuery, int start,
                            int count, Locale l) {
        return dao.searchItems(searchQuery, start, count, l);
    }
      . . .
}
/* Aspecto */
public aspect CatalogEjbHandler {
      . . .
      pointcut categoryHandler() :
            execution(public * CatalogEJB.*(..));

      after() throwing(CatalogDAOSysException se) throws EJBException :
            categoryHandler() {
            throw new EJBException(se.getMessage());
      }
      . . .
}
```

Cenário 2

Este cenário também é muito adequado ao reuso. Neste caso, existem várias ocorrências do mesmo tratador em diversos métodos de classes distintas, sendo possível movê-lo para um único comportamento transversal. Porém, ao contrário do Cenário 1, neste caso não é possível definir apenas um conjunto de pontos de junção no aspecto porque os pontos de junção estão relacionados a métodos de classes distintas. Além disso, em algumas ocorrências desse tratador pode ser necessário converter a exceção originalmente encontrada pelo tratador em não-verificada (a exceção capturada é a super-classe `Exception`).

O trecho de código apresentado a seguir, extraído do sistema *Java Pet Store*, ilustra esse cenário.

Código Original

```
/* Classe 1 */
public final class XMLDocumentUtils {
      . . .
  public static void serialize(...) throws XMLDocumentException {
      try {
            . . .
      } catch (Exception exception) {
            exception.printStackTrace(System.err);
            throw new XMLDocumentException(exception);
      }
      return;
  }
      . . .
}
/* Classe 2 */
public class TPAInvoiceXDE extends XMLDocumentEditor.DefaultXDE {
      . . .
  public String internalTPAInvoiceXDE(...) throws XMLDocumentException {
      try {
            . . .
      } catch (Exception exception) {
            exception.printStackTrace(System.err);
            throw new XMLDocumentException(exception);
      }
      return;
  }
      . . .
}
```

Código Refatorado

```
/* Classe 1 */
public final class XMLDocumentUtils {
      . . .
  public static void serialize(...) throws XMLDocumentException {
      . . .
      return;
  }
      . . .
}
/* Classe 2 */
public class TPAInvoiceXDE extends XMLDocumentEditor.DefaultXDE {
      . . .
  public String internalTPAInvoiceXDE(...) throws XMLDocumentException {
      . . .
    return  ... ;
  }   . . .
}
/* Aspecto */
public aspect XmlDocumentsHandler {
      . . .
      pointcut serializeHandler() :
            execution(* XMLDocumentUtils.serialize(..));
      pointcut internalTPAInvoiceXDEHandler() :
            execution(* TPAInvoiceXDE.internalTPAInvoiceXDE(..));
      declare soft : TransformerException : serializeHandler();
      . . .
      after() throwing(Exception exception) throws XMLDocumentException :
            serializeHandler() ||
            . . .
            internalTPAInvoiceXDEHandler() {
            exception.printStackTrace(System.err);
            throw new XMLDocumentException(exception);
      }
      . . .
}
```

Cenário 3

Este é um cenário que também permite reuso. Foi possível identificar que um mesmo tratador ocorre em classes de pacotes distintos. Como a abordagem de criação dos aspectos é por pacote, nesse estudo de caso do sistema exemplo, não é possível reusar esses tratadores através da abordagem usada nos Cenários 1 e 2. Desta forma, utiliza-se o mecanismo de herança simples entre aspectos para reutilizar o código do comportamento transversal, contornando assim o problema dos tratadores ocorrerem em pacotes distintos. Em seguida, é implementado o aspecto abstrato `XMLDocumentExceptionGenericAspect` e criados os aspectos concretos `XmlDocumentsHandler` e `SupplierpoEjbHandler` (um para cada pacote) para

estendê-lo.

O trecho de código apresentado a seguir, extraído do sistema *Java Pet Store*, ilustra esse cenário.

Código Original

```
/* Classe 1 */
package com.sun.j2ee.blueprints.supplierpo.ejb;
public class SupplierOrder {
      . . .
  public static SupplierOrder fromXML(...) throws XMLDocumentException {
      System.err.println(buffer);
      try {
            . . .
      } catch (XMLDocumentException exception) {
        System.err.println(exception.getRootCause().getMessage());
        throw new XMLDocumentException(exception);
      }
  }
      . . .
}
/* Classe 2 */
package com.sun.j2ee.blueprints.xmldocuments;
public class OrderApproval {
      . . .
   public static OrderApproval fromXML(...) throws XMLDocumentException {
    try {
            . . .
    } catch (XMLDocumentException exception) {
      System.err.println(exception.getRootCause().getMessage());
      throw new XMLDocumentException(exception);
    }
  }
      . . .
}
```

Código Refatorado

```
/* Classe 1 */
package com.sun.j2ee.blueprints.supplierpo.ejb;
public class SupplierOrder {
      . . .
  public static SupplierOrder fromXML(...) throws XMLDocumentException {
           . . .
  }
      . . .
}
/* Classe 2 */
package com.sun.j2ee.blueprints.xmldocuments;
public class OrderApproval {
      . . .
  public static OrderApproval fromXML(...) throws XMLDocumentException {
           . . .
```

```
  }
      . . .
}
/* Super-aspecto */
public abstract aspect XMLDocumentExceptionGenericAspect {
   public abstract pointcutafterWithPrintXMLDocumentExceptionHandler();
   after() throwing(XMLDocumentException exception) throws
XMLDocumentException :
          afterWithPrintXMLDocumentExceptionHandler() {
        System.err.println(exception.getRootCause().getMessage());
        throw new XMLDocumentException(exception);
      }
      . . .
}
/* Aspecto 1 */
public aspect XmlDocumentsHandler extends
XMLDocumentExceptionGenericAspect {
      public pointcut afterWithPrintXMLDocumentExceptionHandler() :
          execution(* OrderApproval OrderApproval.fromXML(...));
      . . .
}
/* Aspecto 2 */
public aspect SupplierpoEjbHandler extends
XMLDocumentExceptionGenericAspect {
      public pointcut afterWithPrintXMLDocumentExceptionHandler() :
            execution(* SupplierOrder SupplierOrder.fromXML(...));
      . . .
}
```

5.2.2 Cenários de Insucesso

Cenário 4

Este é um cenário, ao contrário dos cenários apresentados na seção 5.1.1, que não possibilita o reuso dos tratadores de exceções. Neste caso, existem grandes semelhanças entre os dois tratadores, ou seja, o tratador de exceções precisa ser “aspectizado” utilizando-se um comportamento transversal do tipo *around*, a exceção capturada pelo tratador é a mesma e precisa ser convertida em não-verificada nos dois casos. Porém, não é possível criar um único comportamento transversal para “aspectizar” os dois tratadores por causa de uma pequena diferença: o texto fixo existente no corpo desses tratadores é diferente.

O trecho de código apresentado a seguir, extraído do sistema *Health Watcher*, ilustra esse cenário.

Código Original

```
/* Classe 1 */
public class ServletSearchComplaintData extends HttpServlet {
      . . .
    public void doPost(...) throws ServletException, IOException {
        try {
            . . .
        } catch (ObjectNotFoundException e) {
            out.println(HTMLCode.errorPageQueries(
            "This complaint does not exist"));
        }
    }
      . . .
}
/* Classe 2 */
public class ServletSearchDiseaseData extends HttpServlet {
      . . .
    public void doPost(...) throws ServletException, IOException {
        try {
            . . .
        } catch (ObjectNotFoundException e) {
            out.println(HTMLCode.errorPageQueries(
            "This disease does not exist"));
        }
    }
      . . .
}
```

Código Refatorado

```
/* Classe 1 */
public class ServletSearchComplaintData extends HttpServlet {
      . . .
    public void doPost(...) throws ServletException, IOException {
      . . .
    }
      . . .
}
/* Classe 2 */
public class ServletSearchDiseaseData extends HttpServlet {
      . . .
    public void doPost(...) throws ServletException, IOException {
      . . .
    }
      . . .
}
/* Aspecto */
privileged aspect HealthWatcherServletExceptionHandlingAspect {
      . . .
      pointcut servletSearchComplaintDataDoPostHandler() :
            execution(* ServletSearchComplaintData.doPost(..));
      pointcut servletSearchDiseaseDataDoPostHandler() :
            execution(* ServletSearchDiseaseData.doPost(..));
      declare soft : ObjectNotFoundException :
            servletSearchComplaintDataDoPostHandler() ||
            servletSearchDiseaseDataDoPostHandler();
```

```
      void around() :
          servletSearchComplaintDataDoPostHandler()   {
            try {
                  proceed();
            } catch (ObjectNotFoundException e) {
                printWriter.println(HTMLCode.errorPageQueries(
                  "This complaint does not exist"));
            }
      }
      void around() :
          servletSearchDiseaseDataDoPostHandler()   {
            try {
                  proceed();
            } catch (ObjectNotFoundException e) {
                printWriter.println(HTMLCode.errorPageQueries(
                  "This disease does not exist"));
            }
      }
      . . .
}
```

Cenário 5

Este é cenário é bastante similar ao Cenário 4. Neste caso, os dois tratadores podem ser "aspectizados" com o comportamento transversal do tipo *after*, a exceção capturada pelo tratador é a mesma e precisa ser convertida em não-verificada nos dois casos. Porém, o motivo pelo qual não é possível criar um único comportamento transversal para "aspectizar" os dois tratadores é o fato de que a exceção lançada pelos tratadores é diferente, ou seja, no tratador da classe `ReclamacaoAlteracao` lança a exceção `ReclamacaoNaoAlteradaException` e o tratador de exceções da classe `ReclamacaoConsulta` lança a exceção `ReclamacaoConsultaErroException`. O trecho de código apresentado a seguir, extraído do sistema Telestrada, ilustra esse cenário.

Código Original

```
/* Classe 1 */
public class ReclamacaoAlteracao implements IReclamacaoObj{
      . . .
      public void execute(...) throws IReclamacaoNaoAlteradaException{
        try {
            . . .
        } catch (SQLException e) {
            IReclamacaoNaoAlteradaException ex =
                  new ReclamacaoNaoAlteradaException();
            throw ex;
```

```
        }
      }
      . . .
}
/* Classe 2 */
public class ReclamacaoConsulta implements IReclamacaoObj{
      . . .
      public ReclamacaoDetalhe execute(...)
            throws IReclamacaoConsultaErroException{
        try {
            . . .
        } catch (SQLException e) {
            IReclamacaoConsultaErroException ex =
                  new ReclamacaoConsultaErroException();
            throw ex;
        }
        return ... ;
      }
      . . .
}
```

Código Refatorado

```
/* Classe 1 */
public class ReclamacaoAlteracao implements IReclamacaoObj{
      . . .
      public void execute(...) throws IReclamacaoNaoAlteradaException{
            . . .
      }
      . . .
}
/* Classe 2 */
public class ReclamacaoConsulta implements IReclamacaoObj{
      . . .
      public ReclamacaoDetalhe execute(...)
            throws IReclamacaoConsultaErroException{
            . . .
            return ... ;
      }
      . . .
}
/* Aspecto */
public aspect ReclamacaoMgrHandler {
      . . .
  pointcut reclamacaoAlteracaoExecuteHandler() :
        execution(* ReclamacaoAlteracao.execute(..));
  pointcut reclamacaoConsultaExecuteHandler() :
    execution(* ReclamacaoConsulta.execute(..));
  declare soft : SQLException : reclamacaoAlteracaoExecuteHandler()
    || reclamacaoConsultaExecuteHandler();
  after() throwing(SQLException e)
      throws IReclamacaoNaoAlteradaException :
        reclamacaoAlteracaoExecuteHandler() {
      IReclamacaoNaoAlteradaException ex =
            new ReclamacaoNaoAlteradaException();
      throw ex;
  }
```

```
  after() throwing (SQLException e)
      throws IReclamacaoConsultaErroException :
        reclamacaoConsultaExecuteHandler() {
    IReclamacaoConsultaErroException ex =
      new ReclamacaoConsultaErroException();
    throw ex;
  }
      . . .
}
```

Capítulo 6

CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo sintetiza as contribuições deste trabalho e sugere alguns trabalhos a serem realizados no futuro.

6.1 Conclusões

Este trabalho apresentou um estudo empírico sobre a modularização de tratamento de exceções utilizando POA. O estudo consistiu em mover para aspectos o código responsável pelo tratamento de exceções em três sistemas de informação reais e com código excepcional diversificado. Entre esses sistemas, dois deles são sistemas orientados a objetos e um deles é um sistema orientado a aspectos. Neste último, distribuição, persistência e controle de concorrência são interesses transversais que foram modularizados através de aspectos na sua versão original. Para todos os sistemas escolhidos, tanto na versão original quanto na refatorada, foram coletadas métricas com o objetivo de quantificar atributos de qualidade bem entendidos na comunidade de engenharia de software, como por exemplo, acoplamento e coesão entre módulos. Os resultados obtidos para essas métricas foram então comparados, a fim de analisar algumas vantagens e desvantagens do uso de POA para modularizar o interesse de tratamento de exceções.

As principais contribuições deste trabalho foram:

- ✓ Uma análise de algumas vantagens e desvantagens do uso de POA (a linguagem AspectJ) para modularizar tratamento de exceções em sistemas orientados a objetos e orientados a aspectos. Essa análise é embasada por um estudo empírico que envolveu três sistemas diferentes, construídos por organizações distintas.
- ✓ Um conjunto de cenários que buscam documentar padrões orientados a aspectos para modularizar tratamento de exceções que visam a melhoria da qualidade do código do sistema e um conjunto de cenários que documentam anti-padrões onde o

uso de aspectos sugere não ser uma solução adequada para modularizar tratamento de exceções. Um desenvolvedor familiarizado com esses cenários provavelmente será mais eficiente ao tentar separar através de aspectos os comportamentos normal e excepcional de um sistema.

- ✓ Um conjunto de cenários que indicam como aspectos podem promover o reuso de código de tratadores de exceções. Esses cenários sugerem também o quanto o reuso de código de tratadores depende de fatores diversos e é difícil de obter mesmo em situações aparentemente simples.

Com o objetivo de verificar se a versão refatorada manteve o comportamento inicial, foram realizados vários testes nas versões originais e refatoradas desses sistemas. Esses testes, apesar de não seguirem uma metodologia formal e cobrirem apenas parte da funcionalidade desses sistemas, foram considerados suficientes para este trabalho, já que o seu objetivo principal era analisar a qualidade do código com a refatoração.

6.2 Trabalhos Futuros

Neste trabalho, durante a execução dos experimentos práticos e através da análise dos resultados apresentados pelas métricas de qualidade, foi possível visualizar a importância de algumas linhas de pesquisa a serem seguidas no futuro.

Em primeiro lugar, é necessário definir uma metodologia que auxilie no uso de aspectos para tratamento de exceções. Ela deveria fornecer estratégias para estruturar os aspectos de tratamento de exceções. Além disso, essa metodologia poderia fornecer guias sobre como levar em consideração os aspectos desde as etapas iniciais do processo de desenvolvimento, apresentando orientações de como identificar os cenários vantajosos à modularização do código de tratamento de exceções com aspectos. Desta forma, o desenvolvedor poderia estruturar o sistema desde o início, evitando construir cenários inapropriados ao uso dos aspectos.

Outro trabalho futuro envolve um estudo mais detalhado dos cenários nos quais resultados piores nas versões refatoradas dos sistemas foram conseqüência de limitações da linguagem AspectJ. A partir desse estudo, seria possível sugerir extensões para AspectJ, a fim de

introduzir na linguagem mecanismos lingüísticos mais focados em tratamento de exceções. Uma extensão possível é tornar os mecanismos para designação de pontos de junção de interesse relativos a tratamento de exceções mais expressivos permitindo que pontos de junção relativos a pontos do código onde uma exceção específica é lançada ou encontrada sejam capturados.

Bibliografia

[1] C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena and A. Staa. On the reuse and maintanance of aspect-oriented software: An assessment framework. In Proceedings of the 17th SBES, pages 19–34, October 2003.

[2] F. Cristian. Exception Handling. In: T. Anderson (Ed.) Dependability of Resilient Computers. BSP Professional Books, UK, pp. 68-97, 1989.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), pages 220–242. Springer Verlag LNCS 1241, 1997.

[4] Ramnivas Laddad. AspectJ in Action. Manning, Greenwich, CT, USA, 2003.

[5] A. Garcia, C. Sant'Anna, C. Chavez, V. Silva, C. Lucena and A. Staa. Separation of concerns in multi-agent systems: An empirical study. In C. Lucena, A. Garcia, A. Romanovsky, J. Castro and P. Alencar, editors, Software Engineering for Multi-Agent Systems II, LNCS 2940. Springer-Verlag, February 2004.

[6] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. Staa. Modularizing design patterns with aspects: A quantitative study. In Proceedings of the 4th AOSD, pages 3–14, Chicago, IL, USA, March 2005.

[7] U. Kulesza, C. Sant'Anna, A. Garcia, C. Lucena and A. Staa. Aspectization of distribution and persistence: Quantifying the effects of aop. Submitted to IEEE Software, Special Issue on AOP, May 2005.

[8] P. Tarr, H. Ossher, W. Harrison and S. Sutton. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". Proceedings of the 21st International Conference on Software Engineering, May 1999.

[9] Sun Microsystems. Java 2 platform, enterprise edition, 2004. Available at http://java.sun.com/j2ee (acessada em 01/01/06).

[10] AspectJ Team. AspectJ Programming Guide. World Wide Web, http://www.eclipse.org/aspectj/ (acessada em 01/01/06).

[11] Ramnivas Laddad. I want my AOP!, Part 1: Separate software concerns with aspect-oriented programming. JavaWorld, January 2002. Available at http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html (acessada em 01/01/06).

[12] Ramnivas Laddad. I want my AOP!, Part 2: Learn AspectJ to better understand aspect-oriented programming. JavaWorld, March 2002. Avaliable at http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2.html (acessada em 01/01/06).

[13] Ramnivas Laddad. I want my AOP!, Part 3: Use AspectJ to modularize crosscutting concerns in real-world problems. JavaWorld, April 2002. Avaliable at http://www.javaworld.com/javaworld/jw-04-2002/jw-0412-aspect3.html (acessada em 01/01/06).

[14] Robert W. Sebesta. Concepts of Programming Languages, fifth edition. Chapter 13. Reading, Massachusetts: Addison-Wesley, 2002.

[15] Tigris. aopmetrics home page, 2005. Address: http://aopmetrics.tigris.org (acessada em 01/01/06).

[16] F. C. Filho, C. M. F. Rubira and A. Garcia. A Quantitative Study on the Aspectization of Exception Handling. In ECOOP'2005 Workshop on Exception Handling in Object-Oriented Systems, Glasgow, UK, 2005. Accepted for publication.

[17] A. Garcia, C. Sant'Anna, C. Chavez, V. Silva, C. Lucena and A. Staa. “Agents and Objects: An Empirical Study on Software Engineering”. Technical Report 06-03, Computer Science Department, PUC-Rio, February 2003. Available at ftp://ftp.inf.puc-rio.br/pub/docs/techreports/ (file 03_06_garcia.pdf).

[18] FENTON, N.; PFLEEGER, S. Software Metrics: A Rigorous and Practical Approach. 2.ed. London: PWS, 1997.

[19] SOMMERVILLE, I. Software Engineering, 6.ed. Addison-Wesley, Harlow, England, 2001.

[20] D. L. Parnas. On the criteria to be used in decomposing systems into modules. Commununications of the ACM, 15(12):1053-1058, 1972.

[21] Soares, S.: An Aspect-Oriented Implementation Method. Doctoral Thesis, Federal Univ. of Pernambuco, (2004).

[22] Peter Lee and Tom Anderson. Fault Tolerance: Principles, Techniques, and Tools. 2nd Edition, Springer-Verlag, Wien, 1990.

[23] Westley Weimer and George C. Necula. Finding and Preventing Run-Time Error Handling Mistakes. In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’04), Vancoucer, Canada, October, 2004.

[24] Java BluePrints program for the Enterprise at Java Software, Sun Microsystems. Address: http://java.sun.com/blueprints (acessada em 01/01/06).

[25] Borland TogetherSoft website. URL: http://www.togethersoft.com (acessada em 01/01/06).

[26] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspectoriented programming. In Proceedings of the 22nd ICSE'2000, pages 418–427, Limerick, Ireland, June 2000.

[27] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[28] Enterprise JavaBeans Specification, Version 2.0 (EJB specification). Available at http://java.sun.com/products/ejb (acessada em 01/01/06).

[29] Java Servlet Specification (Servlet specification). Available at http://java.sun.com/products/servlet (acessada em 01/01/06).

[30] JavaServer Pages Specification (JSP specification). Available at http://java.sun.com/products/jsp (acessada em 01/01/06).

[31] Java Message Service Specification (JMS specification). Available at http://java.sun.com/products/jms (acessada em 01/01/06).

[32] JavaMail API Specification (JavaMail specification). Available at http://java.sun.com/products/javamail (acessada em 01/01/06).

[33] Gregor Kiczales, Mira Mezini: Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. ECOOP 2005: 195-213.

[34] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In Proc. of the 17th OOPSLA Conference, pages 161–173, Seattle, Washington, 2002.

[35] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Aspect-Oriented Software Development, pages 21–35. Addison-Wesley, 2005.

[36] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with aspectj. In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), pages 174–190, Seattle, USA, November 2002.

[37] Fernando Castor Filho and Cecília Mary Fischer Rubira. Implementing coordinated exception handling for distributed object-oriented systems with AspectJ. In Proceedings of the VIII Brazilian Symposium on Programming Languages, pages 128–142, Niterói, RJ, Brazil, May 2004.

[38] C. Szyperski. Component Software: Beyond Object-Oriented Programming. ACM Press and Addison-Wesley, New York, NY, second edition, November 2002.

[39] D. Garlan, R. Allen and J. Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. IEEE Software. 12(6):17-26, 1995.

[40] A. Regalado. 10 emerging technologies that will change the world. Technology Review, pages 97–113, January/February 2001.

[41] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[42] Jörg Kienzle and Rachid Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP'02), LNCS 2374, pages 37–61. Springer-Verlag, June 2002.

[43] A. Garcia, C. Chavez, S. Soares, E. Piveta, R. Penteado, V. Camargo, F. Fernandes. 1o. Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos - WASP'2004 - Relatório do Workshop.

[44] STROUSTRUP, B. An overview of C++. In: ACM SIGPLAN Notices. v.23, n.10. October, 1986. pp. 7-18.

[45] SUN MICROSYSTEM. Java 2 Platform, Standard Edition (J2SE). Disponível em http://java.sun.com/j2se/index.jsp (acessada em 01/01/06).

[46] MICROSOFT CORPORATION. The C# language specification. Disponível em http://msdn.microsoft.com/library/en-us/csspec/html/CSharpSpecStart.asp (acessada em 01/01/06).

[47] Reference Manual for the Ada Programming Language
Copyright (C) 1992,1993,1994,1995 Intermelrics, Inc.

[48] EIFFEL SOFTWARE INC. Eiffel Software - The Home of EiffelStudio and Eiffel ENVisioN!. Disponível em http://www.eiffel.com (acessada em 01/01/06).

[49] Lount, P., "A Brief Introduction to Smalltalk." Disponível em http://www.smalltalk.org/articles/article_20040000_11.html (acessada em 01/01/06).