UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

AVALIAÇÃO DE REFATORAÇÃO DE SOFTWARE COM PROGRAMAÇÃO ORIENTADA A ASPECTOS USANDO MÉTRICAS

JOYCE MEIRE DA SILVA FRANÇA

Uberlândia - Minas Gerais
2013

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

JOYCE MEIRE DA SILVA FRANÇA

AVALIAÇÃO DE REFATORAÇÃO DE SOFTWARE COM PROGRAMAÇÃO ORIENTADA A ASPECTOS USANDO MÉTRICAS

Dissertação de Mestrado apresentada à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Engenharia de Software.

Orientador:
Prof. Dr. Michel dos Santos Soares

Uberlândia, Minas Gerais
2013

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Computação a aceitação da dissertação intitulada "**Avaliação de Refatoração de Software com Programação Orientada a Aspectos usando métricas**" por **Joyce Meire da Silva França** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 19 de Fevereiro de 2013

Orientador: _____
Prof. Dr. Michel dos Santos Soares
Universidade Federal de Uberlândia

Banca Examinadora:

_____
Prof. Dr. Marco Tulio de Oliveira Valente
Universidade Federal de Minas Gerais

_____
Prof. Dr. Marcelo de Almeida Maia
Universidade Federal de Uberlândia

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Fevereiro de 2013

Autor:	**Joyce Meire da Silva França**
Título:	**Avaliação de Refatoração de Software com Programação Orientada a Aspectos usando métricas**
Faculdade:	**Faculdade de Computação**
Grau:	**Mestrado**

_____

Autor

Dedicatória

Dedico este trabalho aos meus pais, Eliene e Jorge França, pela dedicação, esforço e os muitos sacrifícios realizados para me oferecer a melhor educação.

Agradecimentos

Agradeço primeiramente a Deus. Ao meu Criador, Salvador e Amigo seja toda minha gratidão pela vida, saúde, sabedoria, conquistas e sonhos realizados.

Agradeço aos meus pais pela dedicação, apoio, amor e carinho.

Agradeço aos meus irmãos, Márcio e Paula, pela amizade e companheirismo.

Agradeço ao João Marcus, meu companheiro, futuro esposo e grande incentivador a persistir na conquista dos meus sonhos.

Agradeço ao meu orientador, Michel dos Santos Soares, por todos os conhecimentos transmitidos, profissionalismo, auxílio e incentivo.

Agradeço ao Caio Augusto R. dos Santos pelo auxílio nas atividades de implementação.

Agradeço a meus colegas de curso pela colaboração durante o mestrado.

Agradeço a todos meus amigos e familiares pelos conselhos e palavras de incentivo.

Agradeço à CAPES pelo apoio financeiro.

Resumo

Código espalhado e entrelaçado afetam o desenvolvimento de software de diversas maneiras, incluindo fraca rastreabilidade, baixa produtividade, problemas com reuso de código, baixa qualidade e maior esforço para manutenção de software. A Programação Orientada a Aspectos (POA) surgiu como proposta para solucionar essas questões através da modularização de interesses transversais com aspectos. Poucos estudos em avaliação empírica dos benefícios do paradigma orientado a aspectos foram publicados. Os resultados apresentados nesses estudos são frequentemente subjetivos, e alguns estudos são não conclusivos. Além disso, a maior parte desses estudos são baseados na implementação de um ou dois interesses transversais em aspectos, e a avaliação é baseada em poucas métricas de software.

Nesse trabalho, uma avaliação da implementação de POA através de métricas de software é proposta. A principal idéia é implementar o maior número possível de interesses transversais como aspectos, com o foco naqueles que não receberam atenção adequada na literatura. Quatro tipos de interesses transversais foram implementados como aspectos, a avaliação de POA foi baseada através de grande quantidade de métricas e também foi realizada uma verificação do impacto da refatoração nos atributos de qualidade. Dois softwares foram usados como estudos de caso. A partir das métricas e das experiências obtidas durante a fase de implementação, análises quantitativas e qualitativas foram produzidas. Os benefícios e malefícios da refatoração com POA foram relatados para que os desenvolvedores avaliem se POA realmente apresenta a melhor solução. Como conclusão desse trabalho, a refatoração de software com POA não é indicada para todos os tipos de interesses transversais. O interesse transversal rastramento é apontado como um caso em que POA é extremamente relevante.

Palavras chave: Programação Orientada a Aspectos, Refatoração, Métricas de Software, Avaliação Empírica.

Abstract

Code scattering and code tangling affect software development in many ways, including poor traceability of requirements, low productivity, poor overall quality and increased efforts for software maintenance. Aspect-oriented programming (AOP) has emerged as a proposal to address these issues through the modularization of crosscutting concerns to aspects. Few studies on empirical evaluation of the benefits of aspect-oriented paradigm were published. Results presented in these studies are frequently subjective, and some studies are non-conclusive. In addition, most of these studies are based on the implementation of only one or two crosscutting concerns into aspects, and the evaluation is based on few software metrics.

In this work, the evaluation of AOP implementation through software metrics is proposed. The main idea is to implement crosscutting concerns as aspects, with focus on those that were not given properly attention in the literature.

Four types of crosscutting concerns are implemented as aspects, the evaluation of POA was based through large quantity of metrics and also was performed a verification of the impact of refactoring on quality attributes. Two softwares were used as case studies. From metrics and experiences during the implementation phase, quantitative and qualitative analyzes were produced. The benefits and detriments of refactoring with POA were reported for developers to assess whether POA presents the best solution. As conclusion of this work, refactoring software with POA is not indicated for all types of crosscutting concerns. The crosscutting concern tracing is indicated as a case in which POA is extremely relevant.

Keywords: Aspect-oriented Programming, Refactoring, Software Metrics, Empirical Evaluation.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A Programação Orientada a Objetos (POO) emergiu com a promessa de melhorar a modularidade e aumentar o nível de abstração para a implementação de software. Apesar do sucesso da POO, os benefícios ainda não são suficientes considerando o fato que algumas questões sobre modularidade ainda não foram resolvidas.

A POO é apropriada para modularizar interesses principais (*core concerns*), mas falha quando precisa modularizar interesses transversais [Filman et al. 2005]. A implementação de interesses transversais em POO produz código espalhado e entrelaçado. O impacto negativo de código espalhado e entrelaçado é que o desenvolvimento de software é afetado de várias maneiras, incluindo fraca rastreabilidade, baixa produtividade, problemas com reuso de código, baixa qualidade e maior esforço para manutenção de software [Laddad 2003].

Em busca de uma forma melhor de modularizar o software, o paradigma de Programação Orientada a Aspectos (POA) [Kiczales et al. 1997] foi proposto no final da década de 1990. A proposta do paradigma POA é lidar com um problema específico de capturar unidades consistentes de um software que as limitações de modelos tradicionais de programação forçam a ser espalhadas em diversas partes do software [Filman et al. 2005]. A construção proposta para implementar interesses transversais em uma unidade separada foi denominada aspecto. A idéia é melhorar a qualidade de software proporcionando melhor modularização e separação de interesses que em outros paradigmas, como POO.

O Desenvolvimento de Software Orientado a Aspectos tem sido aplicado desde seu surgimento com a promessa de melhorar a modularização tratando interesses transversais. Estudos foram publicados com o foco na avaliação do paradigma POA com relação ao paradigma POO [Ceccato e Tonella 2004] [Madeyski e Szala 2007] [Coelho et al. 2008] [Hanenberg et al. 2009]. Em alguns desses estudos existem afirmações a favor e contra POA quando comparado com POO. Baseado na revisão sistemática que será detalhada no capítulo 3, existem poucos estudos empíricos com foco em avaliar POA, e os resultados dos estudos empíricos não são conclusivos em relação ao uso de POA. Em geral, poucas medidas são usadas e poucos interesses transversais são implementados como aspecto.

Uma forma de avaliar POA é usar métricas de softwares. Métricas de software são valores númericos que podem ser utilizados para previsões e margens de erros, como também medir a qualidade do código [Koscianski e Soares 2007]. Nesse trabalho, as métricas de software serão utilizadas como uma forma de avaliar POA. O objetivo é avaliar novos casos de refatoração utilizando POA e analisar o impacto na qualidade de software causado pela refatoração com POA.

Refatoração é o processo de melhorar o *design* de um código existente através da alteração da estrutura interna sem afetar seu comportamento externo [Fowler 1999]. Nesse trabalho a refatoração consiste em implementar os interesses transversais contidos no código fonte do software como aspectos.

Um estudo experimental é proposto nesse trabalho. Dois sistemas POO foram refatorados com POA para avaliar qual paradigma proporciona mais benefícios quando avaliados com métricas de software. A refatoração com POA nos dois softwares possibilitou a avaliação de interesses transversais que não foram investigados apropriadamente na literatura. A avaliação entre os paradigmas POO e POA será baseada nas métricas extraídas automaticamente por ferramentas de qualidade de software.

O estudo experimental foi realizado com dois softwares de tamanhos diferentes. A realização da pesquisa com dois estudos de caso possibilita a comparação entre eles.

1.1 Objetivos

O principal objetivo do trabalho é avaliar experimentalmente a utilização da Programação Orientada a Aspectos. A avaliação empírica de POA proposta nesse trabalho enfatiza questões que não foram abordadas suficientemente pela literatura.

Os resultados obtidos a partir do desenvolvimento dos estudos de caso têm a finalidade de contribuir para a formulação de conclusões mais claras sobre a qualidade de software utilizando POA. Os resultados obtidos nos estudos empíricos encontrados na revisão sistemática são confrontados com os resultados obtidos nesse trabalho com o objetivo de produzir conclusões sobre POA fundamentadas em diversos estudos empíricos.

O desenvolvimento dos estudos de caso objetiva:

- Relatar a implementação de interesses transversais pouco explorados como aspectos;
- Avaliar a qualidade do software refatorado com POA baseado nos resultados das métricas de software;
- Apresentar uma análise crítica da refatoração de software com Programação orientada a aspectos.

1.2 Metodologia

O Desenvolvimento de Software Orientado a Aspectos (DSOA) foi proposto com o objetivo de melhorar a qualidade do código promovendo melhor modularização de interesses. Estudos experimentais com o foco em avaliar se DSOA realmente fornece benéficios durante o desenvolvimento foram publicados. Contudo, os estudos publicados não são unânimes em concluir que o paradigma orientado a aspectos produz benefícios significativos. Diferentes conclusões sobre a avaliação de POA foram apresentadas, dentre elas pode-se destacar:

- POA apresenta muitos benefícios, apesar de incovenientes como diminuição de desempenho [Mortensen et al. 2012].
- POA não fornece benefícios reais ou os benefícios são irrelevantes [Madeyski e Szala 2007] [Bartsch e Harrison 2008].
- POA claramente não fornece benefícios [Przybylek 2010] [Przybylek 2011].

A grande divergência entre as conclusões sobre POA exige que mais estudos experimentais sejam produzidos para contribuir com a formação de uma conclusão mais clara dos benefícios da utilização de POA.

O processo de pesquisa iniciou-se com o objetivo de responder questionamentos inerentes à programação orientada a aspectos. Dentre as indagações existentes, pode-se destacar:

1. Qual o impacto da refatoração orientada a aspectos na qualidade do software?
2. Como mensurar e avaliar a qualidade de POA em comparação com outros paradigmas como POO?

A POA foi proposta para melhorar a modularização de software através da melhora na Separação de Interesses [Dijkstra 1997]. Contudo, é possível afirmar que a inserção de aspectos promove mais benefícios do que outros paradigmas existentes? A primeira questão surge com a intenção de descobrir se existem estudos empíricos que apresentem os benefícios de POA sobre a qualidade do código. Além disso, é importante investigar quais são as vantagens e desvantagens obtidas a partir da refatoração de software com POA. Dentro desta questão, o objetivo desse trabalho é analisar os malefícios decorrentes da utilização de POA, e assim, avaliar se benefícios proporcionados pela utilização de POA superam os malefícios. A Engenharia de Software, em geral, recebeu críticas com relação a pequena quantidade de experimentos com validação quantitativa [Shaw 2002] [Runeson e Höst 2009]. Com relação a POA, o cenário não é diferente. Os poucos estudos empíricos publicados não são suficientes para formular conclusões sobre POA.

A segunda questão indaga a maneira como os pesquisadores produzem resultados de comparação de POA com outros paradigmas. A partir desse questionamento almeja-se descobrir de que forma os estudos experimentais formulam conclusões sobre a qualidade de software com POA. Segundo Sommerville [Sommerville 2010], a qualidade de software é avaliada a partir de atributos internos e externos de qualidade de software. Sendo assim, quais atributos internos e externos são considerados para mensurar a qualidade do software com POA?

Uma revisão sistemática da literatura foi desenvolvida para encontrar os estudos com avaliações empíricas sobre POA. O objetivo da revisão sistemática é encontrar como os estudos experimentais avaliam a qualidade do código com POA e como a avaliação é desenvolvida. Além disso, é importante encontrar pontos que ainda não foram explorados suficientemente para responder as questões propostas nesse trabalho. A revisão sistemática direcionou a metodologia desse trabalho e está detalhada no Capítulo 3.

Baseado na revisão sistemática, esse trabalho propõe produzir estudos de casos que implementem interesses transversais que não foram implementados ou foram pouco implementados, como por exemplo, *logging*, rastreamento, persistência, checagem de inicialização de classe, dentre outros que são apresentados na Tabela 3.4 da página 55. Muitos tipos de interesses transversais foram pouco considerados, o que demanda novas investigações.

Outra informação importante observada nos resultados da revisão sistemática foi que outros estudos de casos que aplicam métricas em softwares com POA precisam ser produzidos para que uma conclusão consistente seja formada sobre a utilização de POA. Algumas métricas importantes não foram aplicadas, como por exemplo, métricas com relação a modularidade (Resposta para componentes e Número de filhos de um componente [Chidamber e Kemerer 1994]).

A metodologia de pesquisa utilizada nesse trabalho foi a avaliação empírica de POA. A aceitação de estudos empíricos em Engenharia de Software e suas contribuições para aumentar os conhecimentos têm crescido continuamente. Isso ocorre porque pesquisas analíticas não são suficientes para investigação de assuntos reais e complexos [Runeson e Höst 2009].

Dois estudos de caso foram usados como instrumento de pesquisa [Yin 2003]. Os estudos de caso foram planejados para dar ênfase em pontos pouco explorados nos estudos empíricos encontrados na revisão da literatura. O critério mais importante para a escolha do estudo de caso foi a possibilidade de usar POA em várias partes do código fonte para garantir uma conclusão concreta sobre os benefícios deste paradigma. A escolha para estudo de caso foi baseada em um conjunto de critérios previamente estabelecidos, descrito a seguir:

- Software que possui interesses transversais para serem transformados em aspectos;
- Software com código fonte livre para download;

- Software codificado na linguagem orientada a objetos Java;
- Dois softwares de tamanhos diferentes;
- Software sem erros de execuções.

Respeitando os critérios de escolha citados anteriormente, dois softwares foram escolhidos como estudo de caso: ATM [ATM] e JMoney [JMo]. O ATM (*Automated Teller Machine*) é um software que simula um caixa bancário eletrônico. O JMoney é um sistema de gerenciamento financeiro pessoal. Nos capítulos 4 e 5 são detalhados os procedimentos realizados para a escolha do ATM e JMoney, respectivamente.

Os softwares escolhidos passaram por um processo de refatoração, modelado no diagrama de atividades da Figura 1.1. O processo de refatoração basicamente consiste na criação de uma nova versão do software. Essa nova versão será construída utilizando uma linguagem de POA chamada AspectJ. As duas versões do software, POO e POA, serão comparadas para avaliar qual paradigma fornece mais benefícios para o desenvolvimento do software. A comparação será baseada em análises quantitativas e qualitativas. As análises quantitativas serão fundamentadas a partir dos resultados obtidos por um conjunto de métricas de software. As análises qualitativas serão formuladas a partir das experiências obtidas durante a implementação dos aspectos.

A primeira etapa da refatoração consistiu na identificação dos tipos de interesses transversais existentes no software. O foco do estudo é implementar o maior número possível de interesses transversais como aspecto com o objetivo de investigar a influência de cada aspecto no código. Dessa forma, todos os interesses transversais identificados no código do software serão implementados em unidades de modularização na nova versão POA criada.

Em seguida, uma nova versão do software foi criada. Os interesses transversais encontrados foram implementados como aspectos utilizando a linguagem de Programação Orientada a Aspectos AspectJ.

Após a fase de implementação, o software refatorado foi testado com a finalidade de verificar se a execução da nova versão não apresenta erros e se o comportamento e as funcionalidades permaneceram inalteradas em comparação com o software original.

Na etapa seguinte, o software escolhido foi submetido a duas ferramentas de qualidade que extraem automaticamente métricas de software. As duas ferramentas utilizadas foram: Plugin Metrics for Eclipse e Sonar. Os resultados das métricas para as duas versões foram dispostos em tabelas para a comparação.

Finalmente, a última etapa foi comparar as duas versões, POO e POA. A comparação foi baseada nos resultados das métricas obtidas para cada versão. O principal objetivo da realização desse estudo experimental é avaliar se a utilização de POA proporciona melhorias no desenvolvimento de software. Através dos resultados das métricas será possível avaliar se POA melhorou a qualidade do código em comparação com POO.

O diagrama de atividades da Figura 1.1 foi modelado para fornecer uma visão geral do processo de refatoração. As atividades desempenhadas na refatoração serão melhor detalhadas posteriormente nos capítulos 4 e 5.

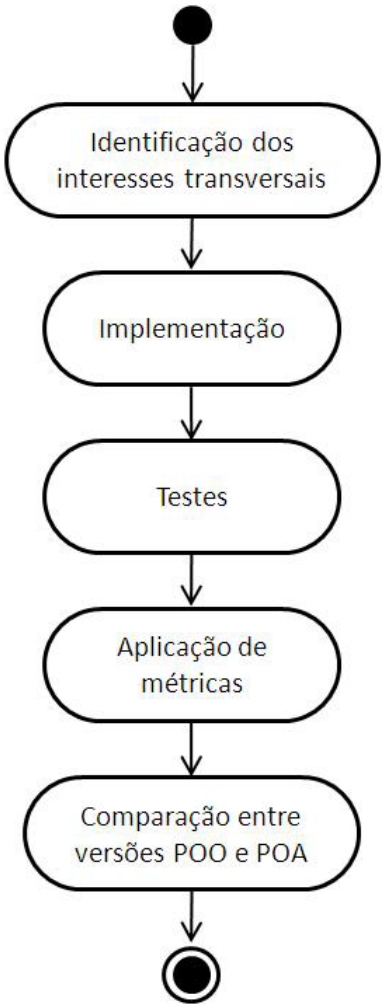

Figura 1.1: Diagrama de atividades para refatoração

Os resultados de pesquisa para estudos experimentais em Engenharia de Software são respostas para avaliações e comparações específicas [Shaw 2002]. Nesse estudo experimental, espera-se responder as questões de pesquisa propostas incluindo contribuir para melhor formulação sobre conclusões que ainda não são consistentes com relação ao uso de POA. Além disso, outro resultado de pesquisa é apresentar um relatório com observações encontradas durante o desenvolvimento de caso, contribuindo com análises qualitativas sobre o uso de POA.

1.3 Organização da dissertação

Esta dissertação está organizada em 6 capítulos. Uma breve descrição do conteúdo de cada capítulo é apresentada a seguir.

Capítulo 2 - Referencial Teórico.

Este capítulo apresenta os fundamentos teóricos da pesquisa.

Primeiramente, os conceitos relacionados com Programação Orientada a Aspectos são definidos. Posteriormente, as definições que fundamentam a linguagem de programação AspectJ, utilizada para refatoração nos estudos de caso, são apresentadas.

Outro assunto abordado nesse capítulo, são as métricas de software que são definidas como instrumento para medir qualidade de software. Duas ferramentas foram utilizadas para extrair automaticamente métricas dos softwares utilizados nos estudos de caso: Plugin Metrics for Eclipse e Sonar. As definições de cada métrica calculada por essas ferramentas são apresentadas nas Seções 2.4 e 2.5.

Capítulo 3 - Revisão Sistemática.

Este capítulo apresenta detalhadamente as etapas desenvolvidas na revisão sistemática da literatura.

A revisão sistemática foi realizada com o objetivo de agrupar os estudos empíricos que avaliam POA que foram publicados em conferências e revistas científicas importantes na Engenharia de Software. Além disso, os resultados da avaliação de POA encontrados em cada artigo são apresentados.

Este capítulo foi baseado no artigo "*A Systematic Review on Evaluation of Aspect Oriented Programming Using Software Metrics*" [França e Soares 2012] publicado nas *proceedings* da *International Conference on Enterprise Information Systems (ICEIS)* em 2012.

Capítulo 4 - Estudo de caso 1: ATM.

Neste capítulo o primeiro estudo de caso é apresentado.

Um software gratuito chamado ATM, originalmente codificado na linguagem de programação Java, é refatorado utilizando POA. Medições foram realizadas nas duas versões do ATM para futuras análises e comparações. Os detalhes da refatoração e os resultados produzidos nesse estudo de caso são apresentados nesse capítulo.

Capítulo 5 - Estudo de caso 2: JMoney.

Este capítulo apresenta os procedimentos realizados para o desenvolvimento do segundo estudo de caso.

O segundo estudo de caso tem por objetivo avaliar a refatoração com POA em um software de tamanho maior que o ATM. Além disso, o JMoney possibilitou a implementação de outros tipos de interesses transversais.

Todas as etapas realizadas (refatoração, extração de métrica e avaliação dos resultados) são detalhadas nesse capítulo.

Capítulo 6 - Conclusão e Trabalhos Futuros.

Este capítulo fornece uma discussão dos resultados encontrados nos dois estudos de caso produzidos, ATM e JMoney. Esses resultados são confrontados com os resultados dos estudos empíricos encontrados na revisão Sistemática para formular conclusões concisas sobre POA.

Os capítulos 4, 5 e 6 foram sintetizados para a produção do artigo "*An Empirical Evaluation of Refactoring Crosscutting Concerns into Aspects using Software Metrics*" aceito para publicação em 2013 nas *proceedings* da *International Conference on Information Technology - New Generations (ITNG).*

Capítulo 2

Referencial Teórico

Neste capítulo são destacados os fundamentos teóricos importantes para o desenvolvimento da pesquisa.

Os conceitos fundamentais da Programação Orientada a Aspectos são apresentados na Seção 2.1. Conceitos relacionados à linguagem de POA AspectJ são apresentados na Seção 2.2. Na Seção 2.3, as métricas de software utilizadas nos estudos de caso são definidas e as ferramentas usadas para extrair as métricas são apresentadas.

2.1 Programação orientada a aspectos

A Programação Orientada a Aspectos (POA) surgiu na década de 1990 [Kiczales et al. 1997] com o objetivo de auxiliar a POO com a dificuldade em lidar com a modularização de interesses transversais. O propósito do paradigma POA é tratar o problema específico de capturar unidades consistentes de um software que as limitações dos modelos de programação tradicionais, como POO, forçam a serem espalhadas em diversas partes do software [Filman et al. 2005] .

A POA afirma melhorar a modularização do software por meio das seguintes vantagens [Pouria Shaker 2005]:

- Melhor rastreabilidade: interesses transversais podem ser facilmente rastreados nos artefatos de software durante o processo de desenvolvimento do software;
- Facilidade de implementar e compreender: autores/leitores de módulos podem focar na implementação/compreensão de um interesse enquanto que o interesse transversal fica encapsulado em um aspecto;
- Reusabilidade dos módulos: módulos são implementados para um único interesse e não apresenta partes da implementação de outro interesse;
- Melhora a evolução: adicionar um interesse transversal é simplesmente uma questão de adicionar um comando quantificado.

Os conceitos fundamentais de POA são apresentados a seguir. Os conceitos que são base de uma linguagem específica, como por exemplo o AspectJ, serão apresentados na seção seguinte.

2.1.1 Separação de Interesses

Separação de Interesses é um princípio de Engenharia de Software [Bourque et al. 2002]. A modularização é considerada uma das melhores maneiras de alcançar Separação de Interesses. Técnicas de modularização têm sido estudadas desde a década de 1960 por diversos autores, como Dijkstra [Dijkstra 1997], Parnas [Parnas 1972] e Wirth [Wirth 1971]. A idéia básica de modularização é dividir o software em componentes menores para tratar a complexidade.

Parnas [Parnas 1972] afirmou que para criar sistemas que são fáceis de implementar, compreender, verificar, e evoluir, é necessário decompor o sistema em módulos de tal forma que cada módulo encapsula um aspecto do sistema que pode ser evoluído independentemente de outros aspectos. Essa decomposição proporciona módulos fracamente acoplados que podem ser implementados, compreendidos, verificados, e evoluidos independentemente. Em outras palavras, os interesses independentes do sistema devem ser identificados e localizados nos módulos. Essa é essencialmente a maneira de alcançar Separação de Interesses [Pouria Shaker 2005].

2.1.2 Interesses transversais

Interesses transversais (*crosscutting concerns*) são interesses que ficam espalhados em vários módulos. São interesses que com a Programação Orientada a Objetos são difíceis de serem implementados sem produzir código espalhado [Laddad 2003].

Diversos tipos de interesses transversais e suas definições são apresentados a seguir. Os quatro primeiros tipos foram implementados como aspectos nos estudos de caso desse trabalho.

***Logging*:** é a produção de mensagens específicas da lógica de um determinado pedaço de código [Laddad 2009]. *Logging* são registros informativos das funcionalidades executadas pelo sistema que é destinado a ser visualizado por usuários finais, administradores de sistemas e pessoal de suporte.

Rastreamento (*Tracing*): é comumente considerado como a produção de mensagens para eventos de baixo nível, como entrada e saída de métodos, construção de objetos, tratamento de exceções e modificação de estado. Rastreamento são registros de informação dos métodos utilizados durante a execução do sistema que é destinado a ser utilizado por engenheiros de serviço ou desenvolvedores.

Checagem de Pontos Nulos: são comandos de código responsáveis por impedir o acesso a variáveis nulas, e assim evitar erros durante a execução de código.

Tratamento de Exceção: Uma exceção é um evento que ocorre durante a execução de um programa rompendo o fluxo normal das instruções executadas pelo programa [Arnold et al. 2006] [Deitel e Deitel 2007].

Quando ocorre um erro em um método, este cria um objeto e o entrega para o sistema de execução. O objeto, chamado objeto de exceção, contém informações sobre o erro, incluindo seu tipo e o estado do programa quando o erro ocorreu. A criação do objeto de exceção e a entrega para o sistema de execução é chamado de lançar uma exceção.

Após o lançamento de uma exceção pelo método, o sistema de execução tenta encontrar algo para tratá-la. O sistema de execução procura na pilha de chamadas (*call stack*) por um método que contém um bloco de código (em Java, são os blocos *try-catch*) que trata a exceção. Esse bloco de código é chamado de tratador de exceção. A procura é iniciada com o método no qual o erro ocorreu e continua através da pilha de chamadas. Se o sistema de execução exaustivamente procura em todos os métodos na pilha de chamadas sem encontrar um tratador de exceções apropriado, o sistema de execução (e consequentemente, o programa) termina.

O tratamento de exceção é um interesse importante que é implementado com a finalidade de evitar paradas bruscas nos softwares por erros não tratados. O tratamento de exceção é considerado um interesse transversal porque sua implementação fica espalhada em várias partes do código. Em Java, os blocos *try-catch* ficam espalhados em diversos métodos das classes.

Segurança: Aplicações de software frequentemente permitem o acesso de dados a usuários. Esse acesso deve ser realizado de maneira segura. A segurança é constituída de muitos componentes, tais como autenticação, autorização, auditoria, proteção contra ataques de web site, e criptografia.

A implementação de segurança utilizando técnicas de programação convencionais exigem a modificação de múltiplos módulos para adicionar códido de autenticação e autorização. A autenticação é um processo que verifica a identidade do usuário para permitir o seu *login* no software. Para implementar controle de acesso no software é necessário invocar o código referente a segurança em vários métodos e módulos [Laddad 2009]. A autorização é um processo que determina se um usuário autenticado possui privilégio suficiente para acessar um determinado recurso. Por exemplo, somente usuários com privilégios de administrador podem acessar determinados dados e funcionalidades. A implementação de uma boa estratégia de autorização evita que

a aplicação fique vulnerável e evita a divulgação de informações, adulteração de dados e elevação de privilégios à usuários não autorizados [Int 2013].

Persistência: Persistência de dados é o processo responsável por armazenar dados permanentemente no computador [Deitel e Deitel 2007]. Os dados podem ser mantidos em dispositivos de memória secondária como discos rígidos e fitas magnéticas, mas também em banco de dados. As aplicações reais frequentemente realizam a persistência de seus dados em banco de dados.

A linguagem de programação Java fornece uma biblioteca (*JPA - Java Persistence API*) para facilitar o mapeamento Objeto-Relacional [Keith e Schincariol 2009]. A *API* de Persistência usa uma abordagem objeto-relacional de mapeamento para preencher a lacuna entre um modelo orientado a objeto e um banco de dados relacional. O *JPA* pode ser usado em aplicações Java SE e Java EE. *Java Persistence* consiste das seguintes áreas: o *Java Persistence API*, a linguagem de consulta e os metadados do relacionamento Objeto-relacional.

Caching: *Caching* é utilizado para melhorar o desempenho do software mantendo o seu comportamento funcional. Um exemplo de estratégia comum utilizada para a realização de *caching* é armazenar na memória rápida do computador dados que são processados com grande frequência para agilizar operações que possuem alto custo de processamento [Laddad 2009].

Caching deve ser utilizado para otimizar pesquisas de dados, evitar constantes requisições de sinais de rede e evitar processamento desnecessário e duplicado. A implementação de *Caching* deve estabelecer quando os dados serão carregados para a memória cache e quando os dados expirados serão removidos da mesma [Int 2013].

Gerenciamento de Transação: Para exemplificar a importância do interesse Gerenciamento de Transação, pode-se considerar como exemplo um comércio eletrônico. Quando um item é adicionado ao carrinho de compras do usuário, o item é removido da lista de estoque. Caso a operação falhe depois que o item é adicionado ao carrinho mas antes de ser removido do estoque, o sistema fica em um estado inconsistente. O sistema contabiliza o item no carrinho de compras e no estoque. Para prevenir essa situação indesejada, é necessário executar as duas operações em uma transação.

Uma transação define uma unidade de trabalho que garante que o sistema mantém um estado consistente antes e depois da sua execução [Laddad 2009]. Se qualquer operação com uma transação falhar, então todo o restante falha, deixando o sistema como estava antes da transação ser iniciada.

Gerenciamento de Transação é um interesse transversal amplamente utilizado em aplicações empresariais [Laddad 2009]. Grande parte das aplicações empresariais utilizam armazenamento com persistência ou um sistema de mensagens que

conduzem à necessidade de gerenciamento de transações. Implementações tradicionais dessa funcionalidade transversal exigem incluir código de gerenciamento de transação em diversos métodos.

O Gerenciamento de transação possui quatro propriedades: atomicidade, consistência, isolamento e durabilidade [Richards 2006].

Atomicidade: o sistema deve executar a transação por completo (*commit*) ou em caso de falhas executar procedimentos de recuperação de falhas (*roll back*) para manter o sistema no estado anterior ao início da transação.

Consistência: a transação não pode transgredir qualquer restrição de integridade do sistema. Por exemplo, incluir em um banco de dados dois registros com chaves primárias iguais. Isso significa que sempre que uma inserção, alteração e remoção é realizada em um banco de dados durante a transação as restrições de integridade são aplicadas e em caso de violações a transação não é concluída com sucesso (*commited*).

Isolamento: propriedade que se refere ao grau de isolamento de execução da transação. A execução de uma transação não deve ser influenciada pela execução de outras transações.

Durabilidade: os resultados de uma transação que termina com sucesso devem permanecer inalterados no banco de dados até que outra transação os altere e também termine com sucesso. Os resultados de transações que terminam com sucesso devem sobreviver a falhas (de transação, de sistema ou de meio de armazenamento).

Controle de Concorrência: O controle de concorrência é a atividade de coordenar a ação de processos que operam em paralelo [Deitel e Deitel 2007].

Um exemplo de aplicação do uso de programação concorrente é o *download* de um arquivo grande (uma imagem ou um vídeo) da Internet. O usuário pode não querer esperar até o *download* completo para iniciar a visualização. Para resolver esse problema, mútiplas *threads* podem ser utilizadas, uma para realizar o *download* e outra para a visualização.

Linguagens de programação que suportam *multithreadings*, como Java, permitem a execução de atividades concorrentes. Java permite que as aplicações possuam execução de *threads* separadas, onde cada *thread* possui sua própria pilha de chamada de métodos e contador de programa. Dessa forma, uma *thread* pode executar paralelamente com outras *threads* compartilando recursos, como a memória [Deitel e Deitel 2007].

Sincronização: Sincronização coordena o acesso a dados compartilhados por múltiplas *threads* [Deitel e Deitel 2007].

Quando múltiplas *threads* compartilham um objeto e esse objeto é modificado por uma ou mais *threads*, resultados indeterminados podem ocorrer, a menos que o acesso do objeto compartilhado seja gerenciado adequadamente. Se uma *thread* está no processo de alterar um objeto compartilhado, outras *threads* não podem ter acesso a esse objeto. A *thread* que está modificando o objeto recebe uma permição de acesso exclusivo e as *threads* que desejam manipular o objeto devem esperar a finalização. A sincronização de *threads* coordena essas situações [Deitel e Deitel 2007].

2.1.3 Aspectos

Assim como uma classe, um aspecto é uma unidade de modularização, encapsulamento e abstração. Embora classes e aspectos tenham muitas similaridades, também há algumas importantes diferenças. Diferentemente das classes, os aspectos podem ser usados para implementar interesses transversais de forma modular [Colyer et al. 2005].

2.1.4 Código espalhado

Código espalhado (*Code scattering*) é causado quando uma única funcionalidade é implementada em múltiplos módulos [Laddad 2009]. Os interesses transversais, por definição, são distribuídos por muitos módulos.

Um sistema bancário que implementa segurança usando técnicas tradicionais é representado na Figura 2.1. Mesmo que seja utilizado um módulo de segurança bem projetado que fornece uma *API* abstrata e esconde os detalhes, cada cliente - o módulo de contabilidade, o módulo do ATM e o módulo do banco de dados - ainda é necessário que o código invoque a *API* de segurança para checar permissão de acesso. O código para checagem de permissão é espalhado em múltiplos módulos, e não há um único lugar para identificar esse interesse. O efeito global é um indesejado entrelaçamento entre os módulos que devem ser protegidos e o módulo de segurança.

2.1.5 Código entrelaçado

Segundo [Laddad 2009], código entrelaçado (*Code tangling*) é causado quando um módulo é implementado para manipular múltiplos interesses simultaneamente. Frequentemente, desenvolvedores consideram interesses como regra de negócios, desempenho, sincronização, *logging*, persistência, segurança e diversos outros interesses quando implementam um módulo. Isso provoca a presença de vários elementos de cada interesse na implementação e resulta em código entrelaçado.

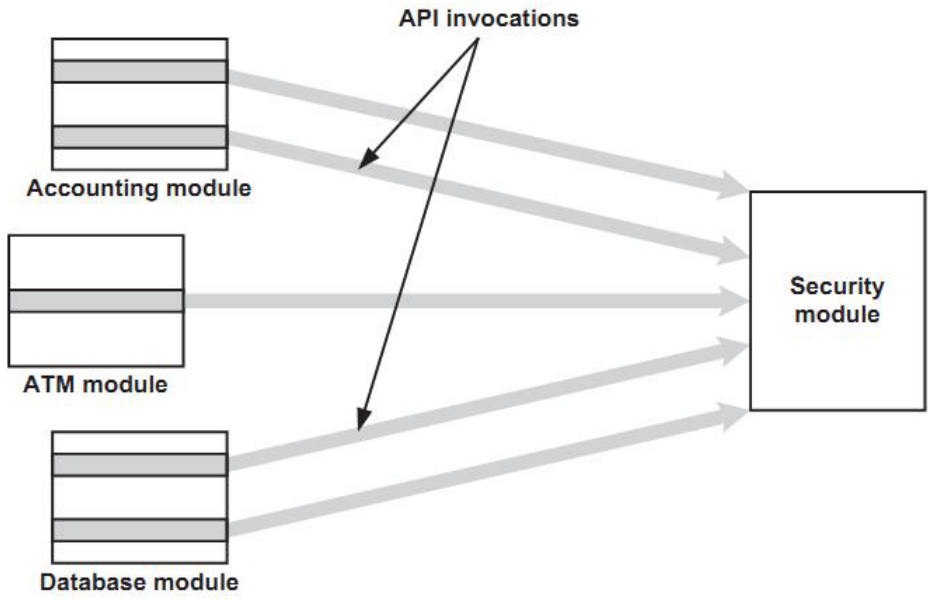

Figura 2.1: Código entrelaçado [Laddad 2009]

Na Figura 2.2, o entrelaçamento de código em um módulo causado pela implementação de vários interesses simultaneamente é ilustrado. Conforme pode ser observado na figura, o código que implementa a lógica de negócios fica misturado com trechos de código relacionados com segurança e gerenciamento de transação.

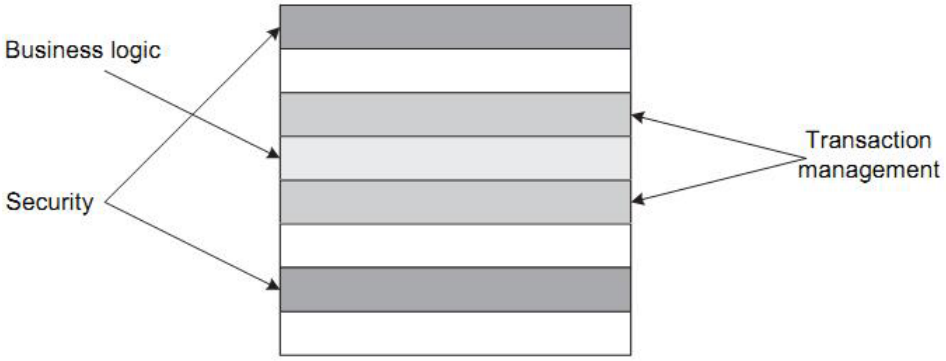

Figura 2.2: Código entrelaçado [Laddad 2009]

2.2 AspectJ

AspectJ é uma extensão de propósito geral orientada a aspectos para a linguagem de programação Java. Dado que AspectJ é uma extensão para Java, todo programa válido em Java é também um programa válido em AspectJ. O compilador AspectJ produz arquivos (classes) que obedecem a especificação do byte-code do Java, permitindo que a máquina virtual do Java execute esses arquivos (classes) [Laddad 2003].

AspectJ é a original e ainda é a melhor implementação de POA [Laddad 2009]. Após algumas *releases* iniciais, a *Xerox* transferiu o projeto AspectJ para a comunidade *open source* eclipse.org.

Nas suas primeiras implementações, o AspectJ estendia Java através de palavras-chave adicionais para suportar conceitos da POA. A versão mais recente do AspectJ, AspectJ 5, inclui suporte para o uso de *annotations*. *Annotations* é um padrão para tipos de assinaturas que especifica metadados (dados ou informações adicionais) sobre o elemento anotado [Laddad 2009]. Por exemplo, um tipo anotado como @Entity indica que esse tipo é para persistência.

Para a fase de implementação dos aspectos foi utilizado um *plugin* AspectJ para Eclipse: AJDT [AJD]. O projeto AJDT (*AspectJ Development Tools*) é uma ferramenta para platorma Eclipse que fornece suporte para Desenvolvimento Orientado a Aspectos com AspectJ.

Os exemplos contidos nesse capítulo são baseados em AspectJ5, porque foi a versão utilizada nos estudos de caso. O uso da versão do AspectJ com *annotations* e não com palavras resevadas como *aspect* e *pointcut* foi determinado a partir do fato de que a ferramenta Sonar não consegue extrair métricas a partir de código com aspectos. O primeiro estudo de caso realiazado, o ATM, foi refatorado usando o AspectJ com as palavras *aspect* e *pointcut* e as métricas do Sonar não foram medidas adequadamente. A partir do uso do AspectJ com *annotations* o Sonar considerou o aspecto como uma classe qualquer e conseguir realizar a extração das métricas efetivamente.

Os conceitos importantes do AspectJ são definidos a seguir.

2.2.1 *Join points*

Join points são pontos específicos que podem ser identificados na execução de um programa [Filman et al. 2005]. O *join point* especifica quando o programa deve ser interceptado e logo em seguida, a execução do código do aspecto.

São exemplos de *join points*:

- chamada de método;
- execução de método;
- acesso a um atributo da classe para leitura ou escrita;
- comandos de controle de fluxo (*if*, *for*, *while*).

O AspectJ deliberadamente disponibiliza apenas um subconjunto de *join points*. Por exemplo, o AspectJ expõe *join points* para chamadas de métodos e acesso à atributos, mas não para comandos de controle de fluxo.

Após a identificação de *join points* úteis para uma determinada funcionalidade transversal, é necessário selecioná-los usando uma construção chamada *pointcut*.

```
69    public void updateUI() {
70        super.updateUI();
71        categoryTree.setCellRenderer(new CategoryTreeCellRenderer());
72    }
```

Figura 2.3: Exemplo de *join point*

Um exemplo de *join point* é mostrado na Figura 2.3. Dois *join points* são capturados no trecho de código da Figura 2.3 (sinalizados pelas setas a esquerda): a execução do método *updateUI*(linha 69) e o acesso à variável *categoryTree*.

2.2.2 *Pointcuts*

Os *pointcuts* são responsáveis por capturar os *join points* de acordo com um conjunto de critérios. Em Java, todos os elementos do programa têm assinaturas. Os *pointcuts* utilizam padrões para essas assinaturas para especificar os *join points* que devem ser capturados.

O *pointcut* apresentado na Figura 2.4 captura a execução de qualquer método no código do software, inclusive construtores. Os símbolos "*" e ".." são "curingas" que selecionam os *join points* independentemente do nome do método, tipo de retorno, modificadores de acesso e parâmetros. O primeiro *execution* do pointcut da Figura 2.4 possui três "*" que significam que o *join point* é interceptado independetemente dos modificadores de acesso (*private*, *protected*, *public* ou *static*), do tipo de retorno e nome do método, respectivamente. O segundo *execution* é exclusivo para interceptar a execução de construtores e o "*" representa que o construtor de todas as classes devem ser selecionados. O símbolo ".." é um curinga que sempre vai representar que o *join point* será capturado independente do nome, tipo e quantidade dos parâmetros.

```
20    @Before("execution(* *.*(..)) || execution (*.new(..))")
```

Figura 2.4: Exemplo de *pointcut*

Os pointcuts também são capazes de expor o contexto dos *join points* que capturam. O contexto do *join point* são informações de tempo de execução associadas ao objeto em execução e os argumentos do método. Por exemplo, uma chamada de método é invocada por um objeto e tem associada argumentos e *annotations*. O objeto no qual o método foi invocado, os argumentos e as *annotations* são o contexto do *join point*. O *pointcut* fornece acesso a essas informações para manipulação de informações em tempo de execução.

2.2.3 *Advices*

Os *pointcuts* selecionam os *join points*, mas não executam algum tipo de ação. Para implementar o comportamento transversal, uma construção chamada *advice* é utilizada. O *advice* especifica qual ação deve ser executada no *join point*. Dessa forma, o *advice*

utiliza *pointcut* (para selecionar os *join points*) e código (para ser executado em cada um dos *join points* selecionados).

Os *advices* podem ser de 3 tipos diferentes: *before*, *after* e *around*, a serem executados "antes", "depois", ou "no lugar" da execução de um ou mais joinpoints, respectivamente. O advice do tipo *around* pode ler as informações do contexto e também pode alterá-las, inclusive pode decidir se o *joinpoint* original deve ser executado ou não.

A declaração do *advice* pode conter parâmetros cujos valores podem ser refenciados no corpo do *advice*. Os valores dos parâmetros são fornecidos pelo *pointcut*. *Advices* que são executados "no lugar" de *joinpoints* são capazes de substituir o código a ser executado. Dessa forma, o *advice* do tipo *around* pode executar ações especificadas no aspecto e não executar o código definido no *join point*.

Um exemplo de *advice* é apresentado na Figura 2.5. O *advice* está anotado com @Before sinalizando que ele é do tipo *before* e executa antes do *join point* especificado no *pointcut*. O *pointcut* é definido dentro da *annotation* @Before (linha 20). Conforme pode ser observado na linha 21 da Figura 2.5, o *advice* é denominado *beforeTrace* e possui um parâmetro do tipo *JoinPoint.StaticPart* que captura o contexto estático do *join point* (informações que não se modificam com a execução do sistema). As linhas 22-24 da Figura 2.5 apresentam o corpo do *advice* onde estão todas ações que serão realizadas assim que o *join point* for interceptado.

```
20    @Before("execution(* *.*(..)) || execution (*.new(..))")
21    public void beforeTrace(JoinPoint.StaticPart thisJoinPointStaticPart){
22        if(logger.isLoggable(Level.INFO)){
23            Signature sig = thisJoinPointStaticPart.getSignature();
24            logger.logp(Level.INFO, sig.getDeclaringType().getName(), sig.getName(), "begin ");
25        }
26    }
```

Figura 2.5: Exemplo de *advice*

Exemplos de *advices* do tipo *after* e *around* serão apresentados e explicados posteriormente no detalhamento da implementação dos estudos de caso nos capítulos 4 e 5.

2.2.4 *Aspecto*

O aspecto é a unidade central do AspectJ da mesma forma que a classe é a unidade central em Java. Aspectos são unidades de modularidade pelo qual AspectJ implementa interesses transversais [Filman et al. 2005]. O aspecto pode conter dados, métodos, e classes aninhadas. É a unidade onde são definidos *pointcuts* e *advices*.

Na Figura 2.6 um exemplo de aspecto é apresentado. Conforme pode ser observado na Figura 2.6, aparentemente um aspecto é muito parecido com uma classe, pois possui identificação de *class* (linha 12), atributos (linha 14), construtor (linhas 16-18) e métodos (linhas 20-34). As diferenças que tornam o código da Figura 2.6 um aspecto e não uma classe comum são as *annotations*. Na linha 11, a classe é anotada com @Aspect e os

```
1  package net.sf.jmoney.aspectos;
2
3  import java.util.logging.Level;
10
11 @Aspect
12 public class AspectTrace {
13
14 private Logger logger= Logger.getLogger("trace"); //obtain the logger object
15
16     AspectTrace(){
17         logger.setLevel(Level.ALL); //initializing the log level
18     }
19
20     @Before("execution(* *.*(..)) || execution (*.new(..))")
21     public void beforeTrace(JoinPoint.StaticPart thisJoinPointStaticPart){
22         if(logger.isLoggable(Level.INFO)){
23             Signature sig = thisJoinPointStaticPart.getSignature();
24             logger.logp(Level.INFO, sig.getDeclaringType().getName(), sig.getName(), "begin ");
25         }
26     }
27
28     @After("execution(* *.*(..)) || execution (*.new(..)) ")
29     public void afterTrace(JoinPoint.StaticPart thisJoinPointStaticPart){
30         if(logger.isLoggable(Level.INFO)){
31             Signature sig = thisJoinPointStaticPart.getSignature();
32             logger.logp(Level.INFO, sig.getDeclaringType().getName(), sig.getName(), "end ");
33         }
34     }
```

Figura 2.6: Exemplo de aspecto

métodos são anotados com @Before (linha 20) e @After (lina 28). No AspectJ5, os aspectos, *pointcuts* e *advices* são definidos seguindo a sintaxe apresentada na Figura 2.6.

2.2.5 *Weaving*

Weaving é o processo de composição entre módulos de funcionalidades principais (classes) com aspectos. *Weaving* expressa como o sistema entrelaça a execução do código base e aspectos [Filman et al. 2005].

O AspectJ propõe três modelos de *weaving* [Laddad 2009]:

Source Weaving: O *weaver* é parte do compilador. A entrada para o *weaver* é constituída por classes e aspectos na forma de código-fonte. O *weaver* funciona de maneira similar ao compilador, processando o código e produzindo o *byte code* "costurado" ao aspecto. O *byte code* produzido pelo compliador é compatível com a especificação Java, e qualquer máquina virtual padrão pode executá-lo.

Binary Weaving: A entrada do *weaver* - classes e aspectos - é no formato *byte code*. A entrada *byte code* é compilada separadamente usando o compilador Java ou o compilador AspectJ.

Load-Time Weaving: A entrada usa classes binárias e aspectos binários, inclusive aspectos e configurações definidas no formato XML.

Independemente do modelo de *weaving* utilizado, a execução do sistema é idêntica.

2.2.6 *Weaver*

Weaver é o processador que realiza o *weaving*, também chamado de compilador de POA. O *Weaver* realiza transformações nos elementos de POA de tal forma que eles sejam compatíveis com as especificações do Java, e assim possam ser executados como um programa na máquina virtual do Java. Por exemplo, os aspectos são mapeados em classes, cada dado e método se torna um membro da classe que representa o aspecto.

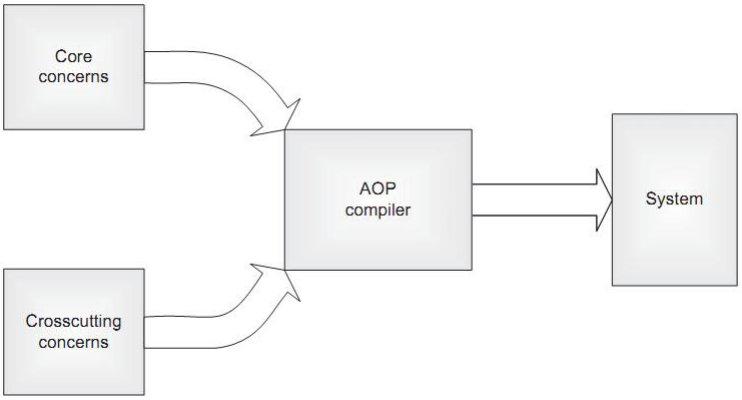

Figura 2.7: Funcionamento do compilador de POA - *Weaver* [Laddad 2009]

O funcionameto do Weaver é ilustrado na Figura 2.7. O compilador de POA recebe como entrada a implementação dos interesses principais e transversais e os entrelaça para formar o sistema final.

2.2.7 AJDT

AJDT [AJD] [Colyer et al. 2005] (*AspectJ Development Tools*) é um *plugin* para IDE Eclipse que fornece suporte para AspectJ. O AJDT auxilia no desenvolvimento de programas POA mostrando os relacionamentos entre os *advices* e os pontos que são alcançados por esses *advices*. O AJDT também fornece suporte para depuração de progamas.

O AJDT foi utilizado para o desenvolvimento dos estudos de caso desse trabalho. Através do AJDT é possível converter um projeto Java existente para um projeto AspectJ, programar utilizando a sintaxe do AspectJ, executar e depurar programas em AspectJ [Colyer et al. 2005].

2.3 Métricas de Software

Por natureza, a engenharia é uma disciplina quantitativa. Engenheiros usam números para ajudá-los a projetar e a avaliar o produto a ser construído. Métricas ajudam engenheiros de software a ganhar profundidade na visão sobre o projeto e da construção do software [Pressman 2011].

A medição de software se ocupa em obter um valor numérico para alguns atributos de software. É possível tirar conclusões sobre a qualidade do software a partir das medições. Uma métrica de software é qualquer tipo de medição que se refira a um sistema de software [Sommerville 2010].

Os fatores que afetam a qualidade de software podem ser classificados em dois grupos: os fatores que podem ser diretamente mensurados são denominados atributos internos de qualidade (por exemplo linhas de código), e fatores que só podem ser medidos indiretamente, que são chamados de atributos externos de qualidade, como por exemplo manutenibilidade.

É difícil medir os atributos externos de qualidade de software diretamente. Atributos como facilidade de manutenção, complexidade e facilidade de compreensão são afetados por muitos fatores diferentes, e não existem métricas diretas e simples para eles. Em vez disso, deve ser usado algum atributo interno (como seu tamanho) e supor que exista uma relação entre o que pode ser medido e o que se precisa saber [Sommerville 2010].

As métricas se dividem em duas classes: dinâmicas e estáticas [Sommerville 2010]. As métricas dinâmicas são coletadas por medições feitas de um programa em execução. As métricas estáticas são coletadas por medições feitas das representações do sistema (projeto, programa, documentação). Métricas dinâmicas ajudam a avaliar a eficiência e a confiabilidade. As métricas estáticas ajudam a avaliar a complexidade, a facilidade de compreensão e a facilidade de manutenção.

Nesse trabalho são utilizadas as métricas estáticas porque são as métricas comumente utilizadas para avaliar qualidade de software nos estudos experimentais [Ali et al. 2010].

As métricas utilizadas nos estudos experimentais encontrados na Revisão Sistemática (3) são dispostas na Tabela 2.1. Uma breve descrição das métricas também é apresentada. Elas foram agrupadas em propriedades de acordo com as suas finalidades. As propriedades abordam tamanho de código, complexidade, coesão, acoplamento, separação de interesses e impacto de mudanças.

As métricas apresentadas na Tabela 2.1 também são utilizadas nos estudos de caso desse trabalho, com exceção das métricas sobre Impacto de Mudanças e Separação de Interesses. As métricas sobre Impacto de Mudança não foram utilizadas porque os estudos desse trabalho não consideraram outras versões do software. Sendo assim, as métricas sobre Impacto de Mudança não se encaixam na proposta dos estudos de caso. As métricas sobre Separação de Interesses não foram consideradas porque as ferramentas utilizadas não extraem essas métricas. Além dessas métricas, muitas outras métricas importantes foram consideradas nesse trabalho e serão definidas nas próximas seções.

Propriedade	Métrica	Definição
Tamanho do código	Linhas de código	Número total de linhas de código, sem incluir comentários e linhas em branco [Sant'anna et al. 2003].
	Tamanho do Vocabulário	Também conhecida como Número de Componentes, conta o número de components (classes e aspectos) no código [Sant'anna et al. 2003].
	Número de atributos	Número de atributos de cada classe ou aspecto [Sant'anna et al. 2003].
	Número de operações	Número de operações (métodos e advices) [Hoffman e Eugster 2008].
	Operações por componente	Número de operações (métodos e advices) de cada classe ou aspecto e o número de seus parâmetros [Chidamber e Kemerer 1994] [Sant'anna et al. 2003].
Complexidade	Complexidade Ciclomática	Número de casos de testes necessários para testar o método de forma abrangente [McCabe 1976].
Acoplamento	Acoplamento entre Componentes	Número de classes e aspectos, métodos, construtores ou campos que são possivelmente chamados ou acessados por outra classe ou aspecto [Chidamber e Kemerer 1994] [Sant'anna et al. 2003].
	Árvore de profundidade de Herança	Comprimento do maior caminho de uma dada classe ou aspecto até a classe ou aspecto raiz da hierarquia [Chidamber e Kemerer 1994] [Ceccato e Tonella 2004].
	Acoplamento Eferente	Número de tipos em um pacote que dependem de tipos que estão fora desse pacote [Kouskouras et al. 2008].
Coesão	Falta de Coesão nas operações	Número de pares de operações (métodos ou advices) trabalhando em diferentes campos de classes/aspectos menos os pares de operações trabalhando em campos comuns [Chidamber e Kemerer 1994] [Ceccato e Tonella 2004].
Separação de Interesses	Difusão de interesses nos componentes	Número de classes e aspectos cujo principal propósito é contribuir para implementação de um interesse e o número de outras classes e aspectos que os acessam [Sant'anna et al. 2003].
	Difusão de interesses nas operações	Número de métodos e advices cujo principal propósito é contribuir para implementação de um interesse e o número de outros métodos e advices que os acessam [Sant'anna et al. 2003].
	Difusão de interesses nas linhas de código	Número de pontos de transição para cada interesse através das linhas de código. Pontos de transição são pontos no código onde há interesses trocados [Sant'anna et al. 2003].
Impacto de Mudança	Número de componentes adicionados/ alterados/ removidos	Número de elementos (classes e aspectos) adicionados/ alterados/ removidos para satisfazer um requisito de mudança [Greenwood et al. 2007].
	Número de operações adicionadas/ alteradas/ removidas	Número de operações (métodos e advices) adicionadas/ alteradas/ removidas para satisfazer um requisito de mudança [Greenwood et al. 2007].

Tabela 2.1: Definição das Métricas utilizadas na Revisão Sistemática

2.4 Plugin Metrics for Eclipse

O Plugin Metrics for Eclipse [Plu] é uma ferramenta automatizada para extração de métricas em software. Metrics é um plugin para a plataforma de desenvolvimento Eclipse [Ecl] [Clayberg e Rubel 2008] que fornece cálculo de métricas e analisador de dependência. Ele disponibiliza várias métricas com média e desvio padrão, faz detecção de ciclos de pacotes e apresenta as dependências em um visualizador utilizando grafos.

Essa ferramenta foi escolhida pela grande quantidade de métricas disponibilizada. Também foi utilizada por ser um plugin do Eclipse, que é a plataforma de desenvolvimento utilizada para a refatoração dos softwares.

As métricas disponibilizadas pelo Metrics são métricas para avaliar sistemas orientados a objetos. Na documentação do Plugin Metrics não há especificação de suporte para sistemas orientados a aspectos. A ferramenta Metrics considera o aspecto como uma

classe comum e produz os cálculos como se fosse um software em Java. Considerando que AspectJ é uma extensão Java, e portanto um sistema implementado com AspectJ possui grande parte do código, toda a lógica de negócios, codificado usando POO, a ferramenta consegue produzir resultados para as métricas normalmente.

Métrica	**Definição**
Linhas de código	Linhas de código. Não considera linhas brancas e comentários [Henderson-Sellers 1996].
Número de classes	Número de classes contidas no código fonte do software [Henderson-Sellers 1996].
Número de métodos	Número total de métodos contidos nas classes [Henderson-Sellers 1996].
Número de atributos	Número total de atributos contidos nas classes [Henderson-Sellers 1996].
Métodos ponderados por classe	Soma da complexidade ciclomática McCabe de todos os métodos numa classe [Henderson-Sellers 1996].
Falta de Coesão nos métodos [Max]	Uma medida para coesão de uma classe. Calculado utilizando o método Henderson-Sellers [Henderson-Sellers 1996]. Se m(A) é o número de métodos que acessam um atributo A, a média de m(A) para todos os atributos é calculada subtraindo o número de métodos m e dividindo o resultado por (1-m). Um valor baixo indica uma classe coesa e um valor próximo de 1 indica falta de coesão e sugere que a classe seja divida em (sub)classes.
Acoplamento Eferente [Max]	Número de classes dentro de um pacote que dependem de classes de outros pacotes [Martin 1994].
McCabe Complexidade Ciclomática [Max]	Número de fluxo de uma parte do código. Cada vez que uma ramificação acontece (*if*, *for*, *while*, *do*, *case*, *catch* e o operador ternário ?:, bem como os operadores lógicos && e || nas expressões) essa métrica é incrementa com 1 [Henderson-Sellers 1996].
Número de filhos	Número de subclasses diretas de uma classe. Uma classe implementando uma interface conta como um filho direto da interface [Henderson--Sellers 1996].
Árvore de profundidade de Herança [Max]	Distância da classe *Object* da hierarquia de herança [Henderson-Sellers 1996].

Tabela 2.2: Definição de Métricas do Plugin Metrics

O Plugin Metrics produz grande quantidade de métricas. Baseado na pesquisa sistemática realizada (Capítulo 3), somente as métricas relevantes foram consideradas nos estudos de caso. Uma breve descrição sobre a definição de cada métrica utilizada nos estudos de casos é apresentada na Tabela 2.2.

2.5 Sonar

O Sonar [Son] é uma plataforma aberta baseada na web para gerenciamento de qualidade de código. Regras, alertas, limites, exclusões e configurações podem ser configurados *online*. Utilizando um banco de dados, o Sonar permite a comparação das métricas com o histórico de medidas.

Atualmente, Java é a única linguagem construída no núcleo do Sonar. Isso significa que quando a plataforma é instalada, o suporte para Java já está contido no Sonar. Todas as outras linguagens são suportadas através de *plugins*.

O Sonar apresenta uma grande lista de métricas para medir a qualidade do software. A definição das métricas utilizadas nos estudos de caso desse trabalho são apresentadas nas tabelas seguintes. Essas definições estão contidas na documentação do Sonar que está disponível no site.

Tamanho	
Métrica	**Definição**
Linhas físicas	Número total de linhas do código.
Linhas de código	Número de linhas de código físicas subtraindo o número de linhas em branco, o número de linhas de comentário, o número de comentários do arquivo de cabeçalho e linhas de código comentadas.
Pacotes	Número de pacotes contidos no código do software.
Classes	Número de classes incluindo classes aninhadas, interfaces, *enums* e anotações.
Arquivos	Número de arquivos analisados no código do software.
Acessores	Número de métodos *getter* e *setter* usados para ler ou escrever em uma propriedade de classe
Métodos	Número de métodos sem incluir acessores. Um construtor é considerado método.
API pública	Número de classes públicas, métodos públicos e propriedades públicas.
Comandos	Número de declarações como definidos na especificação da linguagem Java, mas sem definições do bloco. O contador de declarações é incrementado cada vez que uma expressão, *if*, *else*, *while*, *do*, *for*, *switch*, *break*, *continue*, *return*, *throw*, *synchronized*, *catch*, *finally* é encontrada.

Tabela 2.3: Definição de Métricas de tamanho do Sonar

As métricas apresentadas na Tabela 2.3 são métricas que medem o tamanho do software. O tamanho do software pode ser definido baseado na quantidade de linhas, pacotes, classes, arquivos, métodos e comandos. Diferentemente da métrica Linhas Físicas, a métrica Linhas de código não contabiliza linhas em branco e comentários.

No desenvolvimento do estudo de caso, duas versões, POO e POA, de um mesmo software serão comparadas. Na comparação dos resultados das métricas, a versão que obtiver maiores valores em número de linhas, classes, métodos e arquivos alcança piores resultados. Maiores valores nessas métricas podem produzir impactos negativos no código, maior possibilidade de erros e maior esforço em testes unitários são exemplos de situações

afetadas por um número grande de número de linhas, classes, métodos e arquivos. Visto que as duas versões fornecem as mesmas funcionalidades, a versão que possuir o código com menor número de linhas, classes, métodos, arquivos e comandos apresenta mais vantagens.

As definições das métricas que medem a complexidade de código do software são apresentadas na Tabela 2.4.

O Número de Complexidade Ciclomática (NCC) é também conhecido como métrica McCabe [McCabe 1976]. Para calcular essa métrica, o Sonar simplesmente conta o número de comandos como *if*, *for* e *while* em um método. Sempre que o fluxo de controle de um método divide, o contador NCC é incrementado por um. Cada método tem um valor mínimo de 1 por padrão, exceto acessores que não são contabilizados como métodos e assim não aumentam a complexidade. Para cada um dos comandos que são palavras-chave Java NCC é incrementado em uma unidade: *if*, *for*, *while*, *case*, *catch*, *throw*, *return*, &&, ||, ?.

Como pode ser observado, *else*, *default* e *finally* não aumentam a complexidade ciclomática. Por outro lado, um método simples com um comando *switch* e um bloco com muitos comandos *case* pode ter um valor NCC muito elevado (o NCC tem o mesmo valor ao converter um bloco *switch* para uma sequência equivalente de instruções *if*).

A Tabela 2.4 contém as definições das métricas que fornecem a complexidade média dos métodos, arquivos e classes. Valores baixos são resultados melhores porque sinalizam código menos complexo e consequentemente de mais fácil entendimento.

Complexidade	
Métrica	**Definição**
Complexidade dos métodos	Média do número de complexidade ciclomática por método.
Complexidade das classes	Média do número de complexidade ciclomática por classe.
Complexidade dos arquivos	Média do número de complexidade ciclomática por arquivo.

Tabela 2.4: Definição Métricas de complexidade Sonar

As métricas do Sonar reponsáveis por avaliarem o *Design* do software são definidas na Tabela 2.5.

Resposta para Classe [Chidamber e Kemerer 1994] mede a complexidade da classe em termos de chamadas de métodos. Para cada classe o contador é incrementado: +1 para cada método e +1 para cada chamada a outro método (*getters* e *setters* não são considerados métodos na contagem da ferramenta Sonar). Resposta para Classe é uma métrica importante para avaliar o *Design* e o nível de acoplamento dos componentes do software. Essa métrica verifica o nível de dependência entre as classes. Altos valores para essa métrica é um resultado ruim, porque quanto mais dependente for uma classe de uma outra classe, mais dificil fica a manutenção do código. Alteração na classe significa que mudanças também devem ser feitas nas suas dependentes.

Design	
Métrica	**Definição**
Resposta para classe	O conjunto resposta de uma classe é um conjunto de métodos que podem ser executados em resposta a uma mensagem recebida por um objeto. A métrica Resposta para classe é simplesmente o número de métodos contidos nesse conjunto.
Falta de coesão de métodos	LCOM4 mede o número de componentes conectados de uma classe. Um componente conectado é um conjunto de métodos e campos relacionados.
Índice de pacotes entrelaçados	Fornece o nível de entrelaçamento dos pacotes. Melhor valor: 0%, significa que não há ciclos. O pior valor: 100%, significa que pacotes são totalmente entrelaçados.
Acoplamento entre objetos	Representa o número de outros tipos que estão acoplados a uma classe ou a uma interface. Essa métrica conta o número de tipos de referências que ocorrem através de chamadas de métodos, parâmetros de métodos, tipos de retorno, exceções lançadas e campos acessados.
Profundidade de herança	Fornece para cada classe uma medida dos níveis de herança da parte superior da hierarquia do objeto. Em Java onde todas as classes herdam um objeto o valor mínimo é 1.

Tabela 2.5: Definição de Métricas de *Design* do Sonar

Para medir a coesão das classes do software o Sonar utiliza a métrica LCOM4 (Lack of Cohesion of Methods) baseada em Hitz & Montazer [Hitz e Montazeri 1995]. LCOM4 é uma métrica para sistemas POO que calcula o número de componentes conectados na classe. Um componente conectado é um conjunto de métodos relacionados. Dois métodos **a** e **b** são relacionados se eles acessam uma mesma váriavel ou **a** chama **b**, ou **b** chama **a**. Os valores obtidos pelo software na métrica LCOM4 podem ser avaliados de 3 formas:

- Se LCOM4 = 1, então a classe é coesa, o que representa uma classe "boa";
- Se LCOM4 $\geq$ 2, um problema é sinalizado. A classe deve ser divida em classes menores;
- Se LCOM4 = 0, então não há métodos na classe. A classe é considerada uma classe "ruim".

Índice de pacotes entrelaçados é calculado com base no nível de dependência cíclica entre os arquivos de diferentes pacotes, ou seja, o arquivo **X** do pacote **a** depende do arquivo **Y** do pacote **b** e vice e versa. O melhor valor é 0 % que significa que não há ciclos.

A ferramenta Sonar, além de disponibilizar valores para atributos internos de qualidade, também apresenta os valores de atributos externos de qualidade: Eficiência, Ma-

Rules categories	
Métrica	**Definição**
Eficiência	Avalia como o software utiliza o tempo e os recursos disponíveis. Também calcula o tempo de resposta e de processamento e o desempenho em taxas de transferências. Além disso, contabiliza a quantidade de recursos utilizada e a duração do uso dos recursos [ISO].
Manutenabilidade	Mede o esforço necessário para diagnosticar deficiências, causas de falhas e identificação de partes a serem modificadas. Também considera o esforço necessário para realizar a modificação, prever os riscos de efeitos inesperados e validação do software modificado [ISO].
Portabilidade	Mensura o nível de adaptação a outro ambiente específico. Também considera o nível de esforço necessário para produzir o softaware em outro ambiente [ISO].
Confiabilidade	Mede a maturidade do software. Também considera a tolerância a falhas do software que representa a habilidade do software em manter o nível de desempenho especificado em caso de falhas e recuperar os dados diretamente afetados [ISO].
Usabilidade	Avalia o esforço dos usuários para aprendizagem do uso correto da aplicação [ISO].

Tabela 2.6: Definição de Métricas de *Rules Categories* Sonar

nutenabilidade, Portabilidade, Confiabilidade e Usabilidade. Segundo Sommerville [Sommerville 2010], atributos externos podem ser medidos de forma indireta. Os atributos internos que podem ser medidos de forma direta são utilizados para formular conclusões a respeito dos atributos externos. O Sonar utiliza um conjunto de métricas para aferir sobre os atributos externos de qualidade do software. A definição de cada atributo externo e o conjunto de métricas utilizados para medir a Eficiência, Manutenabilidade, Portabilidade, Confiabilidade e Usabilidade do software são apresentados na Tabela 2.6.

Regras	
Métrica	**Descrição**
Total violações	Número total de regras violadas.
violações pequenas	Número de violações menores.
violações grandes	Número de violações maiores.
violações de informações	Número de violações de informações.
violações críticas	Número de violações críticas.
violações bloqueantes	Número de violações bloqueantes.
índice de regras cumpridas	Fórmula: 100 - (*weighted-violations* / nloc * 100).

Tabela 2.7: Definição de Métricas de Regras Sonar

As métricas definidas na Tabela 2.7 são verificações realizadas pelo Sonar para identificar violações. O Sonar verifica o código fonte e observa pontenciais problemas como:

- Possíveis *bugs* - declarações *try*, *catch*, *finally* e *switch* vazias;
- Código morto - variáveis locais, parâmetros e métodos privados inutilizados;

- Código *Suboptimal* - Utilização de *String* e *StringBuffer* que representam desperdício;
- Expressões complicadas - declarações desnecessárias em laços do tipo *for* que poderiam ser *while*;
- Código duplicado - código copiado e colado pode significar erros copiados e colados.

As violações podem ser classificadas de acordo com o grau de severidade que representam: pequenas, grandes, informativas, críticas e bloqueantes. As críticas e bloqueantes são as mais graves, pois o código sinalizado com tais violações possui erros que podem parar a execução do software abruptamente.

A fórmula utilizada para calcular o índice de regras cumpridas é: 100 - (*weighted-violations* / nloc * 100). O valor *weighted-violations* utilizado na fórmula é a soma das violações ponderadas de acordo com o peso atribuído para cada tipo de violação. As violações com maior nível de severidade possuem um coeficiente maior associado. O usuário do Sonar que possuir permissões de administrador pode configurar o peso associado a cada regra. O valor *nloc* (*number of lines of code*) é o número de linhas de código.

O Sonar determina um índice sobre a qualidade total do software baseado nos resultados obtidos nas métricas. O valor da qualidade total do software pode ser verificado na Tabela 2.8 - Geral. Durante a etapa de comparação das versões POO e POA dos softwares dos estudos de caso, a versão que apresentar maior valor para qualidade total é a versão que mais possui características de boas práticas de programação segundo o Sonar.

Geral	
Métrica	**Definição**
qualidade total	Métrica que indica o valor da qualidade total do software baseada nos resultados de diversas métricas.

Tabela 2.8: Definição Métricas Gerais Sonar

Capítulo 3

Revisão Sistemática

Uma revisão sistemática [Kitchenham 2004] foi realizada para conhecer as evidências empíricas dos benefícios da refatoração utilizando POA.

Dois principais problemas motivaram esse estudo. O primeiro deles é o fato de que poucos estudos com evidência empírica dos benefícios de Programação Orientada a Aspectos (POA) foram publicados. Além desse primeiro problema, um problema derivado é que os estudos não são conclusivos. Por exemplo, pesquisadores chegaram a conclusões de que POA apresenta muitos benefícios, apesar de incovenientes como diminuição de desempenho [Mortensen et al. 2012]. Outros resultados mostraram que POA não fornece benefícios reais ou os benefícios são irrelevantes [Madeyski e Szala 2007] [Bartsch e Harrison 2008], e existem ainda outras publicações claramente aconselhando contra POA [Przybylek 2010] [Przybylek 2011]. Outra questão é que os benefícios são normalmente apresentados sem medidas explícitas ou os resultados são subjetivos [Ali et al. 2010].

Este capítulo é baseado no artigo "*A Systematic Review on Evaluation of Aspect Oriented Programming Using Software Metrics*" [França e Soares 2012] publicado em 2012 nas *proceedings* da conferência ICEIS (*International Conference on Enterprise Information Systems*). A ICEIS é qualificada pela CAPES com conceito B1.

3.1 Método de Pesquisa

A principal questão que motivou essa pesquisa sistemática é:

Q1 - Qual evidência existe na literatura que demonstra que o desenvolvimento de software orientado a aspectos é benéfico? A resposta para esta questão provavelmente será encontrada em trabalhos que desenvolveram estudos empíricos para analisar os benefícios produzidos por POA.

Uma questão derivada é:

Q2 - Como a evidência de benefícios de POA é medida? A resposta para esta questão está relacionada a quais métricas são aplicadas com o objetivo de avaliar POA, e como POA é comparada com POO.

Com o propósito de responder ambas questões, a proposta é realizar uma revisão sistemática [Kitchenham 2004] com o objetivo de identificar quais tipos de pesquisas são realizadas com POA. A busca foi realizada começando em Janeiro de 2006 até Janeiro de 2012. A revisão sistemática foi iniciada com uma busca nas conferências e revistas científicas de engenharia de software.

As revistas científicas escolhidas foram: JSS (*Journal of Systems and Software*), TOSEM (*ACM Transactions on Software Engineering Methodology*), TSE IEEE (*IEEE Transactions on Software Engineering*), e IST (*Information and Software Technology*). As conferências escolhidas foram: CSMR (*European Conference on Software Maintenance and Reengineering*), ICSE (*International Conference on Software Engineering*), AOSD (*International Conference on Aspect-Oriented Software Development*), ECOOP (*European Conference on Object-Oriented Programming*), e OOPSLA (*International Conference on Object Oriented Programming, Systems, Languages and Application*). Artigos publicados em *workshops* específicos realizados juntamente com estas conferências não foram considerados. Essas revistas e conferências foram escolhidas porque são consideradas algumas das mais importantes na área de engenharia de software [Wong et al. 2011].

A *string* de busca escolhida foi (*"aspect oriented" AND "metrics"*). A *string* de busca não contém a palavra *programming*. A justificativa para tal fato é que alguns artigos utilizam o termo "Desenvolvimento orientado a aspecto" ou "Refatoração orientada a aspecto", e o termo "Programação orientada a aspecto" restringiu artigos relevantes.

A visão geral dos resultados de busca pode ser observada na Tabela 3.1. A quantidade de artigos retornados, pré-selecionados e selecionados de cada veículo, bem como o método de busca são apresentados na Tabela 3.1. O detalhamento do processo para selecionar os artigos é apresentado a seguir.

Veículo	**Artigos Retornados**	**Artigos Preselecionados**	**Artigos Selecionados**	**Método de Busca**	**Fonte de dados**
CSMR	4	3	0	automático	IEEExplorer
ECOOP	5	2	2	manual	-
ICSE	26	6	2	automático	IEEExplorer
AOSD	33	9	4	automático	ACM digital library
OOPSLA	13	0	0	automático	ACM digital library
JSS	40	5	2	automático	ScienceDirect
TOSEM	4	1	0	automático	ACM digital library
TSE	4	1	1	automático	IEEExplorer
IST	32	6	2	automático	ScienceDirect
Total	161	33	13		

Tabela 3.1: Visão geral dos resultados de busca

O critério para incluir um artigo na avaliação foi dividido em 3 etapas (conforme

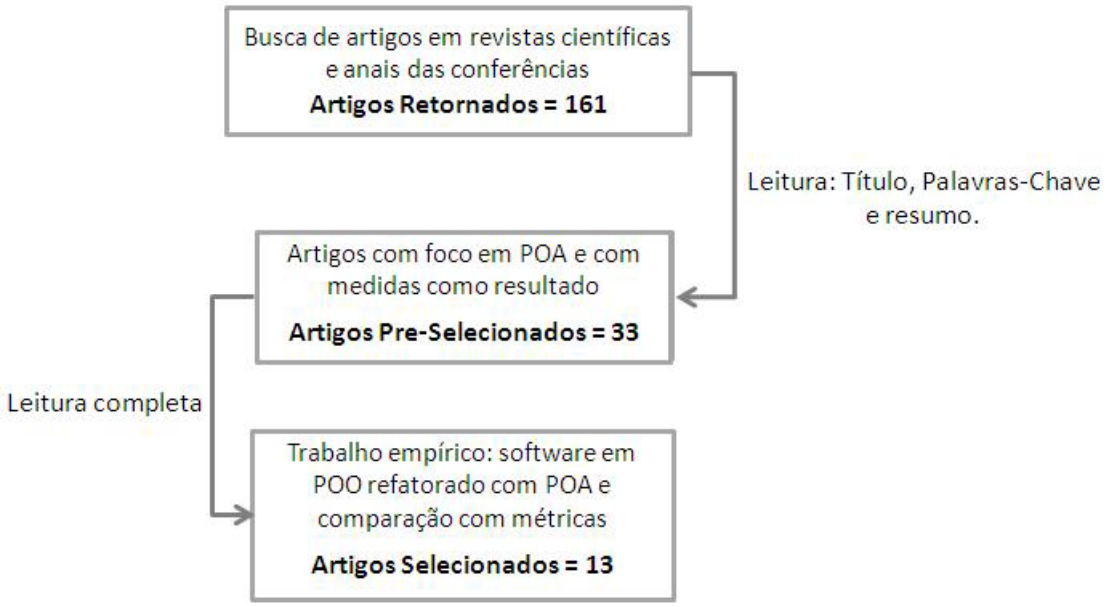

Figura 3.1: Processo para inclusão de artigos

	Veículo/ano	Referência
1	AOSD/06	[Cacho et al. 2006]
2	AOSD/08	[Cacho et al. 2008]
3	AOSD/11	[Ramirez et al. 2011]
4	AOSD/10	[Hovsepyan et al. 2010]
5	ECOOP/08	[Coelho et al. 2008]
6	ECOOP/07	[Greenwood et al. 2007]
7	ICSE/08	[Hoffman e Eugster 2008]
8	ICSE/08	[Figueiredo et al. 2008]
9	IST/08	[Malta e Valente 2009]
10	IST/10	[Tizzei et al. 2011]
11	TSE/10	[Mortensen et al. 2012]
12	JSS/08	[Kouskouras et al. 2008]
13	JSS/11	[d'Amorim e Borba 2010]

Tabela 3.2: Artigos selecionados

ilustrado na Figura 3.1). A primeira etapa foi realizar a busca em cada veículo a partir da *string* ("aspect oriented" AND "metrics"). O número total de artigos retornados a partir da *string* de busca foi 161.

Na segunda etapa, para cada artigo, o título, as palavras-chave e o resumo foram lidos com o objetivo de fazer a seleção. Quando havia dúvida se o artigo deveria ser adicionado, a introdução e a conclusão também eram lidos. Nesta segunda etapa, artigos com foco em AOP como principal preocupação e com alguma medição como resultado foram considerados. Apesar do grande número de artigos retornados, somente uma parte foi completamente lida (33 artigos). A razão disso é porque muitos 'falso positivos' foram retornados. A palavra *aspect* é geralmente utilizada em diversos contextos.

Para a terceira etapa, o critério foi ler completamente os 33 artigos para indentificar

artigos relevantes com foco em uma avaliação experimental. Os artigos relevantes continham aplicação de métricas de software na primeira versão de no mínimo um software, seguido pela identificação de interesses transversais, refatoração com aspectos, seguidos pela aplicação das métricas na versão refatorada do software e a comparação das métricas entre as versões de software (POO e POA).

Como resultado do processo para inclusão de artigos, treze artigos foram escolhidos para análise, como descrito na Tabela 3.2. Com isso, vinte e sete estudos foram considerados. Os outros 20 artigos não foram selecionados principalmente porque, embora apresentassem questões sobre POA, não possuiam o foco em métricas de software, a avaliação foi fraca ou inexistente.

3.2 Critério de Avaliação

Os artigos selecionados foram avaliados baseados em critérios de tipo de avaliação, número de estudos empíricos, propriedades e métricas, e quais interesses transversais foram implementados como aspectos, como descrito nessa seção.

3.2.1 Tipo de Avaliação

A avaliação de cada um dos artigos selecionados foi classificada em: ambiente industrial real, experimento controlado e estudo de caso. A classificação foi definida segundo a definição considerada em cada artigo. Nesse trabalho, ambiente industrial real significa que a pesquisa foi aplicada na prática em uma empresa. Isso é diferente de um estudo de caso, que envolve trabalhar com um exemplo de aplicação, frequentemente um projeto *open-source*. Um experimento controlado significa que uma abordagem experimental para avaliação foi realizada com sujeitos humanos participando e sendo avaliados enquanto desempenham tarefas.

Outros campos de comparação incluem o número total de estudos apresentados em cada artigo, o nome da(s) aplicação(ções) desenvolvidas e apresentadas no artigo, o tamanho da aplicação dada em LOC (ND significa não disponível), a linguagem de programação original na qual a aplicação foi desenvolvida, e a linguagem de POA usada no processo de refatoração. Alguns dos artigos selecionados apresentaram mais de uma aplicação como estudo. Por isso, na Tabela 3.3 o número de estudos empíricos é apresentado em uma coluna. Quando o artigo tem mais de um estudo, os resultados de cada métrica são comparados com cada estudo. Por exemplo, se o artigo tem dois estudos, e o resultado da métrica for positivo/negativo para ambos, para essa métrica o artigo obteve conclusão positiva/negativa. Contudo, se um estudo do artigo teve resultado positivo e o outro negativo, então esse artigo recebe uma marca de inconclusivo.

A lista dos interesses transversais mais comumente considerados [Filman et al. 2005]

Tipo de Avaliação	Artigo	Número de estudos	Aplicação	Tamanho (LOC)	LP Original	LP Aspecto
Ambiente Industrial	[Ramirez et al. 2011]	1	Plato	1212	Java	AspectJ
	[Mortensen et al. 2012]	3	InstanceDrivers	1600	C++	AspectC++
			PowerAnalyzer	13900		
			ErcChecker	51600		
	[Kouskouras et al. 2008]	1	Telecom	1700	Java	AspectJ
Experimento Controlado	[Hovsepyan et al. 2010]	2	Toll System	363	Java	AspectJ
			Pacemaker	369		
Estudo de caso	[Hovsepyan et al. 2010]	2	Toll System	363	Java	AspectJ
			Pacemaker	369		
	[Cacho et al. 2006]	3	Middleware	ND	Java	AspectJ
			Measurement tool			
			Agent-based			
	[Cacho et al. 2008]	1	MobileMedia	4000	Java ME	AspectJ
	[Coelho et al. 2008]	3	Health Watcher	8825	Java	AspectJ
			Mobile Photo	1571		
			JHotDraw	21027		
	[Greenwood et al. 2007]	1	Health Watcher	4000	Java	AspectJ, CaesarJ
	[Hoffman e Eugster 2008]	3	Telestrada	3400	Java	AspectJ
			Pet Store J2EE	17800		
			Health Watcher	4000		
	[Figueiredo et al. 2008]	2	MobileMedia	3000	Java	AspectJ
			BestLab	10000		
	[Malta e Valente 2009]	4	Jaccounting	11676	Java	AspectJ
			JHotDraw	40022		
			Prevayler	2418		
			Tomcat	45107		
	[Tizzei et al. 2011]	1	MobileMedia	11000	Java ME	AspectJ
	[d'Amorim e Borba 2010]	2	Health Watcher	5500	Java	AspectJ
			Library System	600		

Tabela 3.3: Sumário de estudos

Artigo	Interesses Transversais transformados em aspectos											
	S	P	C	R	NP	TE	CT	Si	CC	GT	CIC	ED
[Cacho et al. 2006]												
[Cacho et al. 2008]						●						
[Ramirez et al. 2011]												
[Hovsepyan et al. 2010]	●							●				●
[Coelho et al. 2008]		●				●			●	●		●
[Greenwood et al. 2007]		●				●			●			●
[Hoffman e Eugster 2008]						●			●			
[Figueiredo et al. 2008]		●				●						
[Malta e Valente 2009]						●					●	●
[Tizzei et al. 2011]						●						
[Mortensen et al. 2012]			●	●	●	●	●					●
[Kouskouras et al. 2008]												●
[d'Amorim e Borba 2010]		●							●			●
Total	1	4	1	1	1	8	1	1	4	1	1	7

Tabela 3.4: Tipos de Interesses Transversais implementados nos estudos

é apresentada na Tabela 3.4. As siglas usadas nessa tabela significam: S - Segurança, P - Persistência, C - *Caching*, R - Rastreamento, NP - Checagem de Pontos Nulos , TE - Tratamento de Exceção, CT - Configurações em Tempo de Execução, Si - Sincronização, CC - Controle de Concorrência, GT - Gerenciamento de Transação, CIC - Checagem de Inicialização de Classe, ED - Específico de Domínio.

Os tipos de interesses transversais implementados como aspectos em cada artigo são apresentados na Tabela 3.4. Conforme pode ser observado na tabela, dois artigos ([Cacho et al. 2006] e [Ramirez et al. 2011]) não deixaram claro quais tipos de interesses transversais foram refatorados com POA. Outro ponto que pode ser notado na Tabela 3.4 é que a maioria dos estudos implementaram 1,2 ou 3 tipos de interesses transversais como aspectos. Apenas 3 dos treze artigos implementaram 4 ou mais interesses transversais com POA nos estudo experimentais.

Na última linha da Tabela 3.4 pode ser observado o número total de implementação de cada tipo de interesse transversal nos treze artigos. Conforme pode ser observado na Tabela 3.4 um importante interesse transversal *logging* (amplamente citado como exemplo na literatura de POA [Filman et al. 2005] [Laddad 2009]) não foi implementado por nenhum dos 27 estudos. Além disso, oito tipos de interesses transversais foram avaliados por apenas um artigo.

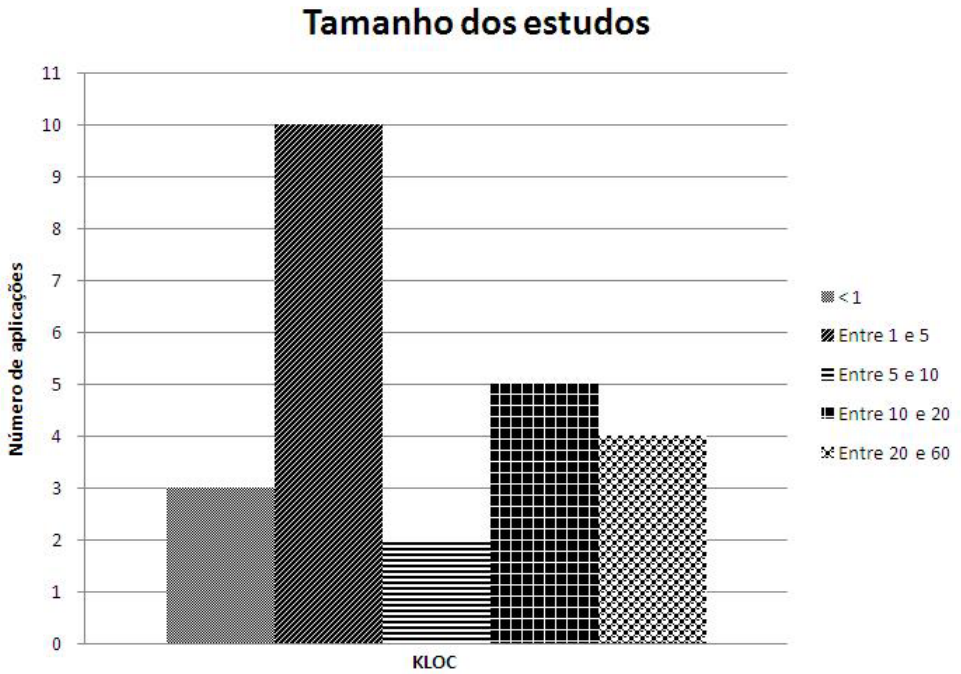

Figura 3.2: Gráfico dos tamanhos dos estudos encontrados na Revisão Sistemática

O gráfico apresentado na Figura 3.2 mostra os tamanhos dos softwares utilizados nos estudos encontrados na Revisão Sistemática. Vinte e sete estudos aplicações foram listadas na Revisão sistemática, contudo no gráfico são expostos o tamnho de vinte e quatro. As três aplicações que não constam no gráfico não tiveram o tamanho especificado no

artigo [Cacho et al. 2006]. Conforme pode ser observado nesse gráfico, 15 aplicações o que representa a grande maioria das aplicações utilizadas nos estudos de caso encontrados na Revisão Sistemática possuem tamanho menor que 10 KLOC. Das 27 aplicações, apenas nove têm tamanho maior ou igual a 10 KLOC.

As métricas foram agrupadas em propriedades. Propriedades como tamanho de código, coesão e acoplamento são úteis para avaliar o projeto de arquitetura escolhido. A lista final de propriedades e métricas é apresentada nas Tabelas 3.5 e 3.6. Por motivos de espaço no cabeçalho das tabelas, C é abreviação de Conclusão. Foram extraídas as métricas usadas em cada artigo. Além disso, conclusões foram obtidas de cada artigo a partir da comparação entre aplicações de POA e as aplicações que não são POA. A conclusão pode ser positiva (+), negativa (-) ou inconclusiva (?). Resultados positivos são obtidos quando para uma determinada métrica a aplicação de POA conquistou melhor desempenho em comparação com POO. Os resultados são negativos quando a aplicação de POA teve um desempenho inferior comparado com a aplicação de outro paradigma. Resultados inconclusivos são baseados em duas possibilidades. As métricas foram aplicadas em duas aplicações diferentes e os resultados são muito similares, ou uma aplicação obteve conclusão positiva e a outra conclusão negativa.

Propriedade	**Métrica**	**Artigos**	**C**	**Total**
Tamanho do código	Linhas de código (LOC)	[Greenwood et al. 2007]	+	2+, 7-, 3?
		[Mortensen et al. 2012]	+	
		[Cacho et al. 2006]	-	
		[Cacho et al. 2008]	-	
		[Ramirez et al. 2011]	-	
		[Coelho et al. 2008]	-	
		[Figueiredo et al. 2008]	-	
		[Malta e Valente 2009]	-	
		[Tizzei et al. 2011]	-	
		[Hovsepyan et al. 2010]	?	
		[d'Amorim e Borba 2010]	?	
		[Hoffman e Eugster 2008]	?	
	Tamanho do Vocabulário	[Cacho et al. 2006]	+	1+, 6-
		[Cacho et al. 2008]	-	
		[Greenwood et al. 2007]	-	
		[Figueiredo et al. 2008]	-	
		[d'Amorim e Borba 2010]	-	
		[Coelho et al. 2008]	-	
		[Tizzei et al. 2011]	-	
	Número de atributos	[Cacho et al. 2006]	+	3+, 1?
		[Ramirez et al. 2011]	+	
		[Hovsepyan et al. 2010]	+	
		[Cacho et al. 2008]	?	
	Números de operações	[Hoffman e Eugster 2008]	-	1-
	Peso das operações por componente	[Cacho et al. 2006]	+	3+, 2-
		[Ramirez et al. 2011]	+	
		[Hovsepyan et al. 2010]	+	
		[Cacho et al. 2008]	-	
		[Greenwood et al. 2007]	-	
Complexidade	Complexidade Ciclomática por componente	[Ramirez et al. 2011]	+	1+

Tabela 3.5: Propriedades e Métricas - 1

Propriedade	Métrica	Artigos	C	Total
Acoplamento	Acoplamento entre componentes	[Hovsepyan et al. 2010]	+	3+, 3-, 1?
		[Greenwood et al. 2007]	+	
		[Figueiredo et al. 2008]	+	
		[Cacho et al. 2006]	-	
		[Cacho et al. 2008]	-	
		[d'Amorim e Borba 2010]	-	
		[Hoffman e Eugster 2008]	?	
	Árvore de Profundidade de Herança	[Ramirez et al. 2011]	+	1+, 2-
		[Cacho et al. 2008]	-	
		[d'Amorim e Borba 2010]	-	
	Acoplamento Eferente	[Ramirez et al. 2011]	+	1+, 2-
		[Tizzei et al. 2011]	-	
		[Kouskouras et al. 2008]	-	
Coesão	Coesão das operações	[Cacho et al. 2006]	+	6+, 3-
		[Ramirez et al. 2011]	+	
		[Hovsepyan et al. 2010]	+	
		[Greenwood et al. 2007]	+	
		[Tizzei et al. 2011]	+	
		[d'Amorim e Borba 2010]	+	
		[Cacho et al. 2008]	-	
		[Hoffman e Eugster 2008]	-	
		[Figueiredo et al. 2008]	-	
Separação de Interesses	Difusão de interesses sobre componentes	[Hovsepyan et al. 2010]	+	3+, 3-
		[Tizzei et al. 2011]	+	
		[d'Amorim e Borba 2010]	+	
		[Cacho et al. 2008]	-	
		[Greenwood et al. 2007]	-	
		[Figueiredo et al. 2008]	-	
	Difusão de interesses sobre operações	[Hovsepyan et al. 2010]	+	3+, 2-
		[Greenwood et al. 2007]	+	
		[Tizzei et al. 2011]	+	
		[Cacho et al. 2008]	-	
		[Figueiredo et al. 2008]	-	
	Difusão de interesses sobre LOC	[Cacho et al. 2008]	+	6+
		[Hovsepyan et al. 2010]	+	
		[Greenwood et al. 2007]	+	
		[Figueiredo et al. 2008]	+	
		[Tizzei et al. 2011]	+	
		[Mortensen et al. 2012]	+	
Impacto de Mudança	Número de componentes adicionados/alterados/removidos	[d'Amorim e Borba 2010]	+	1+, 1-, 2?
		[Tizzei et al. 2011]	-	
		[Greenwood et al. 2007]	?	
		[Figueiredo et al. 2008]	?	
	Número de operações adicionadas/alteradas/removidas	[Greenwood et al. 2007]	+	2+, 1-
		[Figueiredo et al. 2008]	+	
		[Tizzei et al. 2011]	-	

Tabela 3.6: Propriedades e Métricas - 2

3.3 Discussão

Em geral, foi difícil concluir os benefícios de POA em termos da propriedade de tamanho de código. Sete estudos apresentaram resultados negativos com relação à métrica LOC (o número de linhas de código aumentou com POA), 2 positivos e 3 inconclusivos. De acordo com esses resultados, para a maioria dos estudos POA aumenta LOC após a refatoração com aspectos. Este resultado é surpreendente porque era esperado que com melhor modularização com aspectos, e a remoção de linhas de código correspondentes a interesses transversais, o número de LOC deveria diminuir. A métrica tamanho do vocabulário teve resultados negativos em seis artigos, o que significa que o número de elementos como classes, interfaces e arquivos de configuração aumentaram na versão com POA. Isto é natural, considerando que com a introdução de aspectos uma conexão adicional entre

os elementos é necessária.

A métrica número de atributos obteve resultados positivos com a refatoração com POA. Como apenas um artigo mencionou a métrica número de operações, não é possível formular conclusões.

A métrica "peso das operações por componente - *weighted operations per component (WOC)*" foi mencionada em cinco artigos. Três artigos ([Cacho et al. 2006], [Ramirez et al. 2011] e [Hovsepyan et al. 2010]) concluíram que WOC é positivo para POA. Estes três artigos juntos produziram um total de seis estudos empíricos (conforme pode ser observado na Tabela 3.3), então pode-se concluir que em 6 estudos WOC foi positivo para POA. Dois artigos (que somam apenas dois estudos empíricos) obtiveram resultado negativo na métrica WOC. Isto mostra uma tendência de que POA diminui WOC, isto é, diminui a complexidade de um componente. Contudo, análises adicionais são necessárias.

Com apenas um artigo ([Ramirez et al. 2011]), que apresentou somente uma aplicação, não é possível formular conclusões sobre complexidade.

Todas as métricas sobre acoplamento (Árvore de Profundidade de Herança, Acoplamento Eferente e Acoplamento entre componentes) são inconclusivas. Árvore de Profundidade de Herança e Acoplamento Eferente tiveram um resultado positivo e dois negativos cada. Acoplamento entre componentes teve três negativos, três positivos, e um resultado inconclusivo. Baseado nesses resultados, não foi possível determinar que POA fornece qualquer benefício em termos de diminuição de acoplamento.

A Coesão foi investigada em nove artigos, que somam quatorze aplicações. Ela melhorou em oito aplicações, mas foi negativa em seis aplicações. Baseado nisso, embora aparentemente a coesão melhore com POA, não é possível definir conclusões.

Sobre separação de interesses, três métricas foram consideradas: Difusão de Interesses sobre Componentes, Difusão de Interesses sobre operações e Difusão de Interesses sobre LOC (DILOC). Os dois primeiros são inconclusivos, mas a métrica de DILOC foi positiva para todos os seis artigos.

O Impacto de mudanças foi definido a partir de duas métricas. A primeira, 'Número de classes (módulos) adicionadas/modificadas/removidas', é insignificante para dois artigos, e negativa para um artigo. Baseado nisto, aparentemente a aplicação de POA não tem um forte impacto nas classes, porém análises adicionais são necessárias. A segunda, 'Número de operações adicionadas/modificadas/removidas', é positiva (POA faz poucas mudanças) para dois artigos e negativo (POA faz muitas mudanças) para um. Baseado nisto, é difícil chegar em uma conclusão definitiva.

Na Tabela 3.4, é claramente perceptível que a maioria dos interesses transversais implementados foi tratamento de exceção. Portanto, pode ser inferido que aspectos são muito utilizados para modularizar exceções.

Somente três de treze artigos implementaram quatro ou mais interesses transversais, ou seja, a maioria dos pesquisadores não estão introduzindo aspectos em substituição de

muitos interesses transversais. A razão para esta política não ficou clara com a leitura dos artigos, e deve ser investigada em trabalhos futuros.

Após realizar a pesquisa sistemática, a primeira questão de pesquisa: *Qual evidência existe na literatura que mostra que desenvovilmento de software orientado a aspectos é benéfico?* ainda não pôde ser devidamente respondida. Muitos resultados não são conclusivos, porque os estudos apresentam resultados conflitantes.

A única métrica que pode ser considerada conclusiva é a DILOC, pois obteve resultado positivo em todos os estudos que a aplicaram. Seis artigos, que somam 10 estudos (de acordo com as Tabelas 3.6 e 3.3) foram unânimes em afirmar que POA é benéfico para o desenvolvimento de software no sentido que melhora a separação de interesses.

Outro fator que impossibilita a formulação de uma resposta para a questão de pesquisa é que muitos pontos ainda não foram abordados adequadamente. A maioria dos trabalhos implementam menos de três tipos de interesses transversais como aspectos. Considerando que 27 aplicações foram realizadas nesses 13 artigos, o importante interesse de logging não foi implementado. Além disso, na Tabela 3.4 é revelado que a grande maioria dos interesses, 8 de 12 tipos de interesses transversais, foram implementados em apenas 1 artigo. Isso significa que muitos interesses transversais não foram considerados apropriadamente.

A segunda questão de pesquisa: *Como a evidência de benefícios de POA é medida?*, foi respondida principalmente com a aplicação de métricas propostas para programação orientada a objetos. Contudo muitas métricas importantes de POO, tais como 'Resposta para classe/componente' [Chidamber e Kemerer 1994] e 'Número de filhos' [Chidamber e Kemerer 1994], que medem a Modularidade do sistema não foram consideradas. Outro fato encontrado é que poucas métricas específicas para orientação a aspectos foram aplicadas nos estudos.

3.4 Conclusão

Os resultados da revisão sistemática proposta nessa seção são que poucos estudos em evidência empírica dos benefícios do paradigma orientado a aspectos foram publicados, resultados são frequentemente subjetivos, e alguns estudos não são conclusivos. Somente um pequeno número de interesses transversais são considerados pela maioria dos pesquisadores. Além disso, poucas métricas de software são aplicadas, o que torna difícil obter sólidas conclusões, e métricas específicas para POA são aplicadas em poucos estudos. Com relação ao tipo de avaliação, muito mais deve ser feito. Somente dois estudos são desenvolvidos em ambiente industrial real, e somente um experimento controlado foi realizado.

Surpreendentemente, para a maioria dos estudos, o número de linhas de código aumentou. Como a introdução de aspectos aumenta a modularidade, era esperado que o número de linhas de código diminuisse. Também foi surpreendente que as métricas

de software que eram importantes para propriedades arquiteturais, como acoplamento e modularidade, não foram investigadas em nenhum dos artigos analisados.

Como conclusão dessa revisão sistemática, muito mais pode ser feito na avaliação de POA. Um importante resultado é que a introdução de POA em casos reais na indústria ainda está no início, somente cinco de vinte e sete estudos foram realizados em um ambiente industrial real. Experimentos controlados com desenvolvedores também são incomuns (apenas um artigo), o que significa que há um longo caminho a percorrer nesse assunto. Estudos futuros são necessários para compreender o que acontece com a complexidade com a introdução de aspectos no código OO, como apenas um artigo mencionou a métrica complexidade ciclomática.

Capítulo 4

Estudo de Caso 1: ATM

Neste capítulo é proposto um primeiro estudo de caso como forma de avaliar a POA. Resumidamente, a avaliação consiste em refatorar, utilizando POA, um software codificado em POO, extrair métricas de software nas duas versões e avaliar qual versão obteve melhor resultado.

Na Seção 4.1 são apresentados os critérios utilizados para a escolha do primeiro software a ser refatorado, um simulador de ATM. Em seguida, na Seção 4.2, são expostas informações sobre o ATM, como versão, linguagem de programação, tamanho e local onde o código fonte está disponível. Na Seção 4.3 são apresentados os detalhes de implementação da refatoração do ATM utilizando POA. Na Seção 4.4 é mostrado o resultado das métricas aplicadas na versão original do ATM e na versão refatorada. Finalmente, na Seção 4.5, uma análise baseada na comparação dos resultados das métricas apresentados na Seção 4.4.

Os capítulos 4, 5 e 6 foram baseados no artigo "*An Empirical Evaluation of Refactoring Crosscutting Concerns into Aspects using Software Metrics*", aceito para publicação nas *proceedings* da conferência ITNG (*(International Conference on Information Technology - New Generations*) em Abril de 2013. A ITNG é uma conferência qualificada pela CAPES com conceito B1.

4.1 Critérios para escolha do software

O primeiro estudo de caso escolhido foi um simulador de um ATM - *Automated Teller Machine*. O ATM foi usado em outras pesquisas como estudo de caso [Javed et al. 2007] [Cui et al. 2009]. Além disso, outro ponto que motivou a utilização do ATM foi a documentação completa. O acesso à documentação do software diminui o tempo e o esforço da compreensão do código. Diversos softwares de código aberto não possuem documentação disponível, o que torna difícil para desenvolvedores entenderem o funcionamento do código construído por outros desenvolvedores.

Os critérios estabelecidos para a escolha do software e foram atendidos pelo ATM são:

- Software com código fonte livre para download;
- Software codificado na linguagem orientada a objetos Java;
- Software que possui interesses transversais para serem transformados em aspectos;
- Software com código fonte compilando e executando corretamente.

Para o primeiro estudo de caso foi estabelecido que o tamanho não fosse muito grande para que o tempo de apendizagem e implementação da refatoração em aspectos fosse menor. É importante salientar que de acordo com a revisão sistemática apresentada no capítulo 3, dos 27 softwares listados, 13 são menores que 5 KLOC, 11 maiores que 5 KLOC e 3 não mencionaram o tamanho do software. Portanto, o software escolhido como primeiro estudo de caso aparentemente pode ser considerado de tamanho pequeno (menor que 5 KLOC) mas está numa faixa de tamanho considerado pela maioria dos estudos da Revisão Sistemática.

4.2 Apresentação do Software

ATM (*Automated Teller Machine*) [ATM] é a segunda versão de um software que simula um caixa de banco automático. A documentação e o código fonte do ATM estão disponíveis em http://www.math-cs.gordon.edu/courses/cs211/ATMExample/. Com relação a direitos autorais, no site o autor concede livremente permissão para reprodução com propósitos educacionais não-comerciais.

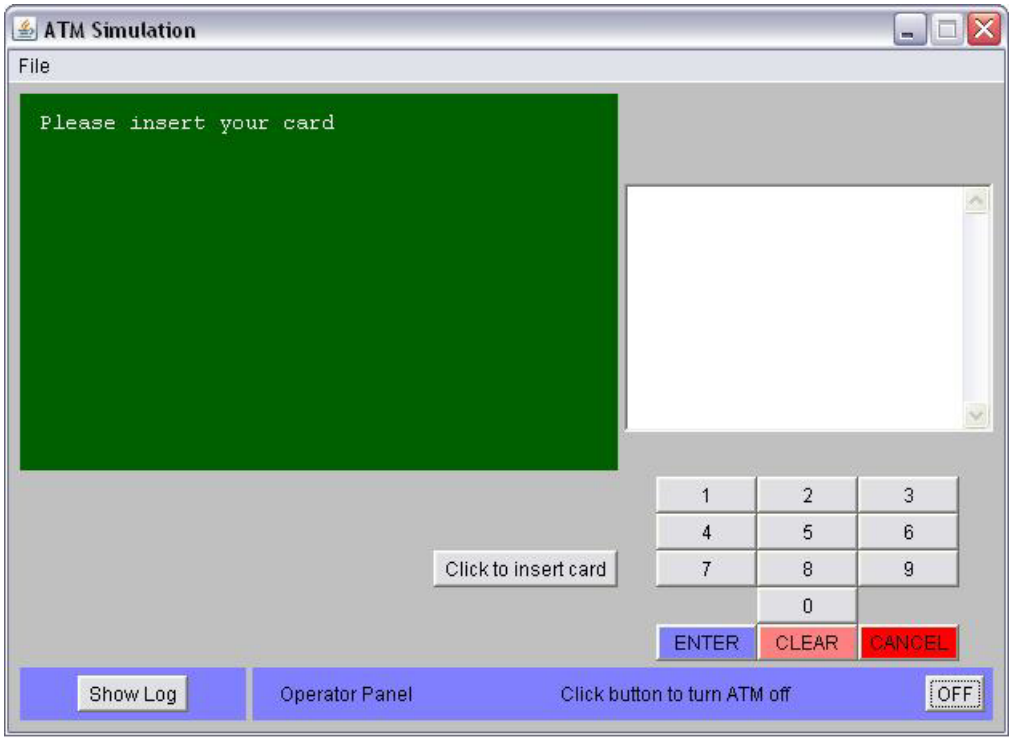

Figura 4.1: Tela do ATM

A tela principal do ATM é exibida na Figura 4.1. O ATM controla um caixa de banco automático com leitor de cartão magnético para ler o cartão, um *console* para cliente (teclado e tela) para interação com o cliente, um *slot* para depositar envelopes, um *dispenser* para notas de dinheiro, uma impressora para imprimir os recibos dos clientes e uma chave/interruptor para permitir que o operador ligue e desligue a máquina.

O tamanho do ATM medido segundo a métrica Linhas de código é aproximadamente 2,5 KLOC. Porém, o código original foi alterado com modificações que serão detalhadas posteriormente e o tamanho passou para a ordem de 3 KLOC.

A documentação disponibilizada pelo autor do ATM é bastante completa e inclui diversos diagramas da UML [Booch et al. 2005], como diagramas de casos de uso e diagramas de sequência. Para auxiliar na compreensão das funcionalidades do ATM são apresentados dois diagramas: o diagrama de casos de uso (Figura 4.2) e o diagrama de classes (Figura 4.3). A documentação completa e os demais diagramas podem ser encontrados no site do ATM.

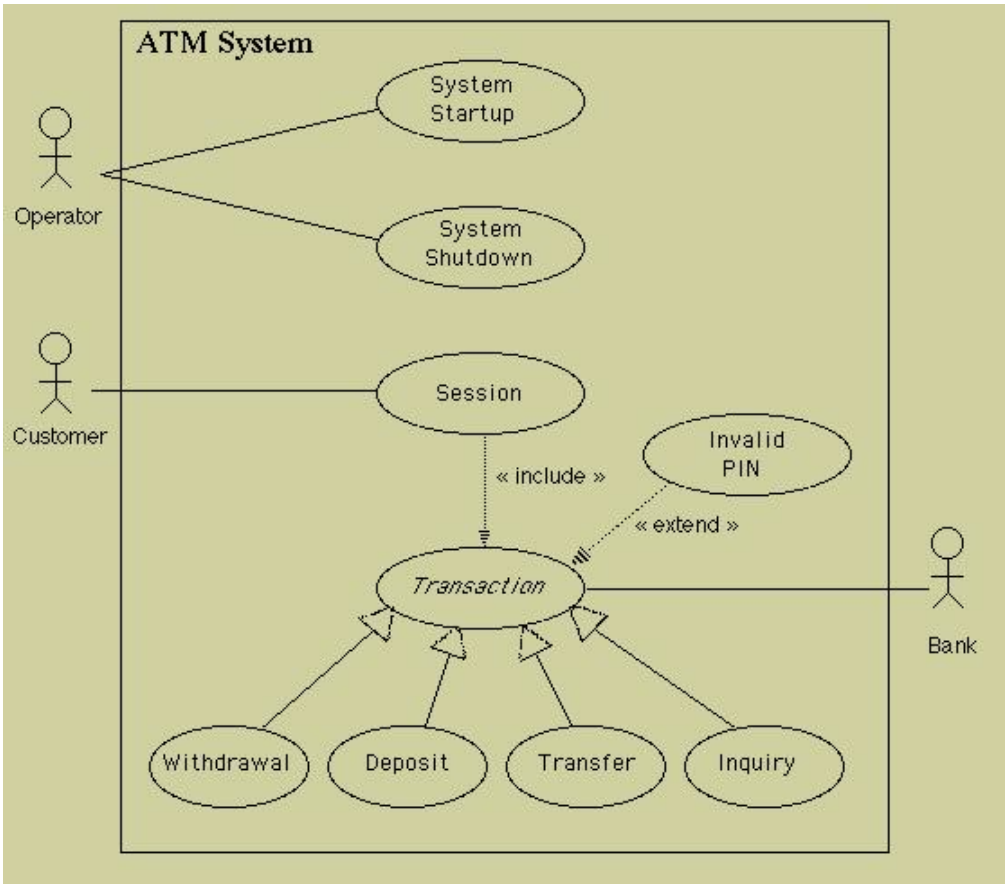

Figura 4.2: Diagrama de casos de uso do ATM

O diagrama de casos de uso apresentado na Figura 4.2 ilustra as formas possíveis de interação com o ATM. Três tipos de atores podem interagir com o sistema: operador, cliente e o banco. O operador é responsável por ligar e desligar o sistema. O cliente pode usar o sistema para realizar transações como saques, transferências, depósitos e consultas.

Para isso, o cliente deve abrir uma sessão e informar a senha válida. O ator que representa o banco também é autorizado a realizar transações.

O diagrama de classes do ATM (Figura 4.3) descreve a estrutura do sistema mostrando as classes e seus relacionamentos. Conforme pode ser observado no diagrama de classes da Figura 4.3, a classe ATM tem um relacionamento de composição com diversas classes do sistema, demonstrando que ATM é composta por várias partes (painel, *console*, impressora, leitor de cartões, *dispenser* de notas, dentre outras).

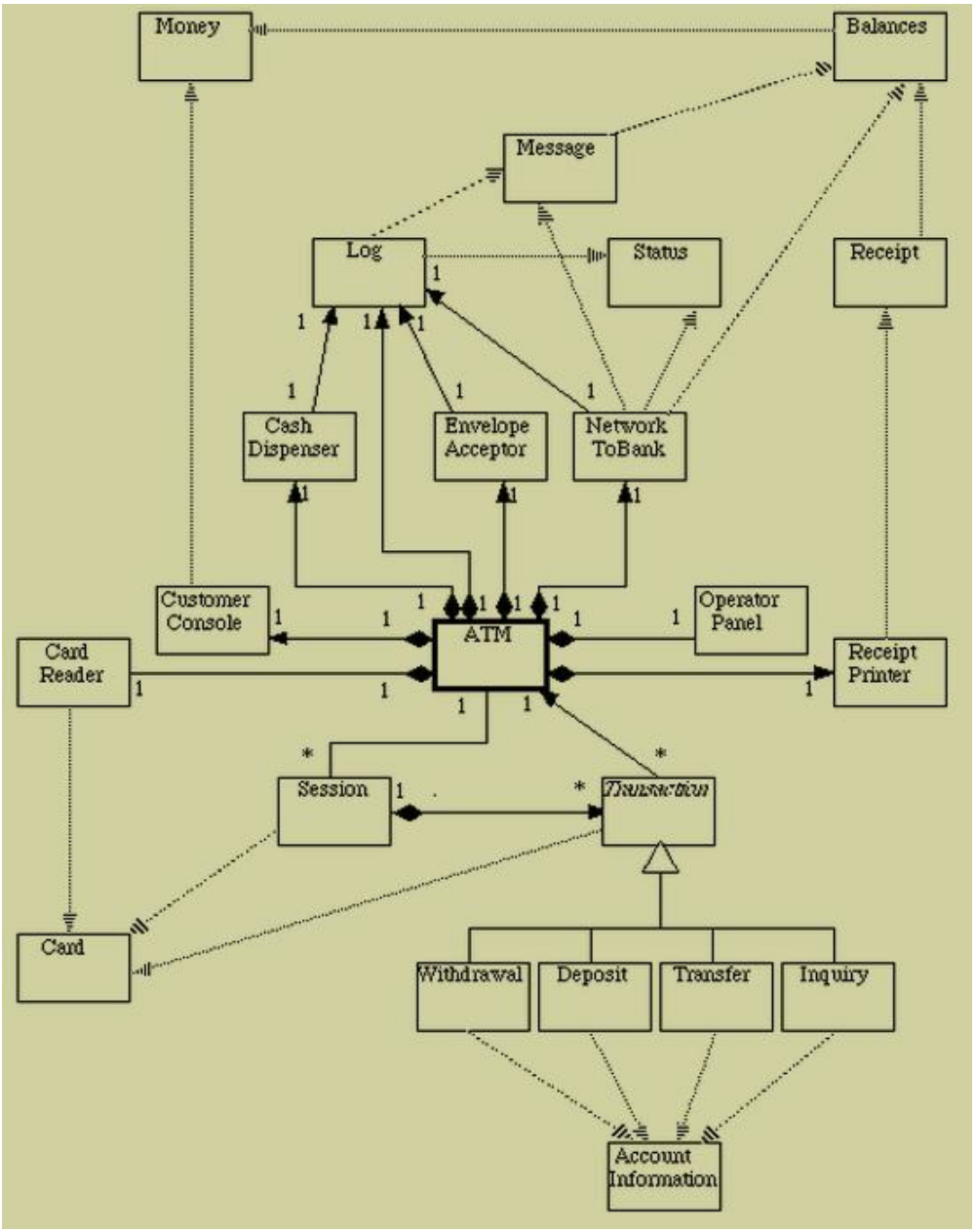

Figura 4.3: Diagrama de classes do ATM

4.3 Refatoração do ATM

A refatoração do ATM com POA foi desenvolvida em etapas, modeladas no diagrama de atividades da Figura 1.1.

O início da refatoração foi marcado pela etapa de identificação dos interesses transversais. Os interesses transversais foram identificados por meio de uma técnica manual.

Os tipos de interesses transversais mais comumente refatorados em aspectos [França e Soares 2012] foram listados, e uma checagem manual no código e na documentação do ATM foi realizada com o objetivo de observar a existência de cada um desses interesses transversais. A lista contém 12 tipos de interesses transversais: *Logging*, Segurança, Persistência, *Caching*, Rastreamento, Checagem de Pontos Nulos , Tratamento de Exceção, Configurações em Tempo de Execução, Sincronização, Controle de Concorrência, Gerenciamento de Transação e Checagem de Inicialização de Classe. Todos os tipos foram observados porque a intenção era encontrar o maior número possível de interesses transversais no código do ATM.

O foco do trabalho é implementar interesses transversais que não foram amplamente utilizados para refatoração com aspectos, baseado nos estudos de casos analisados na revisão sistemática, como por exemplo, *logging*, rastreamento e segurança dentre outros.

Logging foi o primeiro interesse transversal identificado. O diagrama de classes do ATM (Figura 4.3), disponibilizado pelo autor, contém uma classe chamada Log. O código implementado para *logging* do sistema estava espalhado por diversas classes do sistema.

Tratamento de exceção foi outro tipo de interesse transversal encontrado no código do ATM. Blocos *try-catch* ficam espalhados por diversas classes.

Embora o interesse transversal rastreamento não tenha sido encontrado no ATM original, ele foi implementado em aspectos, com o objetivo de implementar um maior número de aspectos possíveis. Além disso, a implementação de rastreamento foi realizada por apenas um estudo de caso na revisão sistemática, o que torna importante a investigação da implementação desse tipo de interesse transversal. Dessa forma, para realizar a comparação das versões Java e AspectJ, o rastreamento também foi implementado na versão Java porém utilizando somente os recursos fornecidos pela linguagem, sem aspectos.

A etapa seguinte no processo de refatoração foi a implementação. O ATM foi transformado de um projeto Java para um projeto AspectJ. Assim, todas as configurações necessárias para o início da implementação dos aspectos foram executadas. Os interesses transversais *Logging*, *Tracing* e Tratamento de Exceções foram refatorados em aspectos utilizando a linguagem de programação orientada a aspectos AspectJ. Outra tarefa realizada nessa etapa foi a implementação em Java do rastreamento na versão POO do ATM.

A etapa relativa aos testes foi realizada com o objetivo de verificar se a versão refatorada do ATM desempenha as mesmas funcionalidades da versão original.

A etapa seguinte no processo de refatoração foi coletar métricas das duas versões do ATM. Para essa etapa foram utilizadas duas ferramentas, Metrics e Sonar, que aplicam as métricas ao sistema e produzem automaticamente os resultados.

A última etapa consistiu em comparar os resultados das métricas coletadas das versões POO e POA do ATM para produzir as análises.

A etapa de implementação dos aspectos será detalhada nas subseções seguintes. As demais etapas (Aplicação das métricas e comparação de resultados) serão detalhadas nas próximas seções desse capítulo.

4.3.1 Implementação de *logging*

O primeiro interesse transversal refatorado como aspectos no ATM foi o *logging*. O requisito *logging* já está implementado no ATM, mas não como aspecto.

Para implementar o requisito *logging* no ATM com aspectos a primeira tarefa foi depurar o código para entender a operação do software. A depuração do código foi muito importante para conhecer as situações onde o *logging* é usado.

A classe Log.java do projeto ATM tem 4 métodos:

- logSend (registra o envio de uma mensagem para o banco)
- logResponse (registra a resposta recebida de uma mensagem)
- logCashDispensed (registra a distribuição de dinheiro pelo terminal do caixa eletrônico)
- logEnvelopeAccepted (registra o aceite de um envelope de depósito)

Esses métodos são chamados em diversos lugares no código. Para implementar logging como aspectos, a implementação e as chamadas desses métodos foram apagadas. *Pointcuts* para acessar os lugares do código onde essas chamadas aconteciam foram criados. Além disso, foram criados *advices* para desempenhar o mesmo trabalho feito dos 4 métodos, ou seja, para registrar situações específicas da aplicação.

Depois que a implementação do aspecto AspectLog foi finalizada e o código em Java que realizava o *logging* foi apagado, a aplicação refatorada foi testada. Os testes foram realizados para checar se a versão refatorada apresenta as mesmas funcionalidades que a versão original. Além de garantir que a nova versão não possui erros, os casos de testes foram criados para verificar se a execução do ATM refatorado é idêntica à do ATM. Como não houve nenhuma diferença no comportamento do ATM refatorado, foi concluído que o ATM refatorado com aspectos apresenta as mesmas funcionalidades do ATM original.

O código da Figura 4.4 apresenta o AspectLog, aspecto criado para registrar as mensagens de log do ATM. A criação de um aspecto utilizando *annotations* no AspectJ se resume em criar uma classe como outra qualquer (palavra *class* na linha 13) e anotá-la

```
1  package aspects;
2  import org.aspectj.lang.annotation.After;
11
12 @Aspect
13 public class AspectLog {
14
15     Log log;
16
17     public AspectLog() {
18         log = new Log();
19     }
20
21     @Before("execution (* atm.physical.NetworkToBank.sendMessage(Message , Balances ))" +
22             " && args (message, balances)")
23     public void beforeLog(Message message, Balances balances) {
24         // Log sending of the message
25         log.logSend(message);
26     }
27
28     @AfterReturning(pointcut="execution (* atm.physical.NetworkToBank.sendMessage(.. ))",
29                     returning="result")
30     public void afterReturningLog(Status result) {
31         // Log the response gotten back
32         log.logResponse(result);
33     }
34
35     @After("execution (* atm.physical.CashDispenser.dispenseCash(Money)) && args(amount)")
36     public void afterLog(Money amount) {
37         log.logCashDispensed(amount);
38     }
39
40     @AfterReturning("execution (* atm.physical.EnvelopeAcceptor.acceptEnvelope(..))")
41     public void afterReturningLog2() {
42         log.logEnvelopeAccepted();
43     }
44 }
```

Figura 4.4: Aspecto criado para fazer o logging do ATM

com @Aspect (linha 12). Um aspecto, assim como classes, pode conter construtor, atributos e métodos. No AspectLog, foi utilizado o atributo log (linha 15) que é inicializado no construtor do AspectLog (linhas 17-19).

O AspectLog possui 4 *advices*. O primeiro deles (linhas 21-26) foi denominado beforeLog e foi anotado com @Before para executar antes da execução do método sendMessage. Além de definir o método a ser interceptado, o *pointcut* também tem acesso ao argumento do método sendMessage e o utiliza para realizar o *logging*. O *advice* afterReturningLog (linhas 28-33) foi anotado com @AfterReturning porque para seguir a lógica original o *logging* só pode ser registrado se o método interceptado for finalizado normalmente e o @AfterReturning garante essa situação. O terceiro *advice* do AspectLog, afterLog (linhas 35-38), foi anotado com @After para executar depois que o método dispenseCash for finalizado. O último *advice*, afterReturningLog2 (linhas 40-43), também foi anotado com @AfterReturning, para garantir que ele seja executado apenas se o método acceptEnvelope retornar com sucesso.

Na Figura 4.5 pode-se observar a diferença no código do método sendMessage da classe NetworkToBank do ATM. Na Figura 4.5(a) o código que implementa o interesse transversal *logging* fica entrelaçado com o código que implementa a lógica de negócios do sistema. Já na Figura 4.5(b) o método ficou mais coeso e com menos linhas, o que torna mais fácil a compreensão da funcionalidade do método. O método não se preocupa em

fazer o *logging*. Essa responsabilidade agora é do aspecto que intercepta esse método e realiza as mesmas tarefas de *logging*, só que agora com melhor separação de interesses.

```
66    /** Send a message to bank
67     *
68     *  @param message the message to send
69     *  @param balances (out) balances in customer's account as reported
70     *         by bank
71     *  @return status code returned by bank
72     */
73    public Status sendMessage(Message message, Balances balances)
74    {
75        logger.log(Level.INFO, "begin " );
76
77        // Log sending of the message
78        log.logSend(message);
79
80        // Simulate the sending of the message - here is where the real code
81        // to actually send the message over the network would go
82
83        Status result = Simulation.getInstance().sendMessage(message, balances);
84
85        // Log the response gotten back
86        log.logResponse(result);
87
88        logger.log(Level.INFO, "end " );
89
90        return result;
91    }
92
93    // Log into which to record messages
94    private Log log;
```

(a) Log sem aspectos do ATM

```
48
49     /** Send a message to bank
50      *
51      *  @param message the message to send
52      *  @param balances (out) balances in customer's account as reported
53      *         by bank
54      *  @return status code returned by bank
55      */
56     public Status sendMessage(Message message, Balances balances)
57     {
58         Status result = Simulation.getInstance().sendMessage(message, balances);
59         return result;
60     }
```

(b) Log com aspectos do ATM refatorado

Figura 4.5: Comparação de código sem aspectos e com aspectos

4.3.2 Implementação de rastreamento

A implementação do requisito rastreamento tem como objetivo produzir, como saída no console, mensagens que sinalizem o início e o fim de todos os métodos no projeto ATM utilizados durante a execução do sistema.

O rastreamento não está implementado no código original do ATM. Portanto, rastreamento será implementado na versão original em Java e na versão refatorada do ATM. A alternativa encontrada para implementar *tracing* no ATM, usando somente Java, foi colocar mensagens de início e fim em todos os métodos do projeto. Essa é uma árdua tarefa, considerando que o ATM possui 166 métodos, incluindo construtores, *getters* e *setters*.

Para realizar rastreamento em Java, mensagens de início foram colocadas como primeiro comando no método, exceto nos construtores que invocam o construtor da super classe, porque é exigido que isso seja o primeiro comando a ser executado. Então, nesses casos, a mensagem é o segundo comando. Mensagens de fim foram colocadas como último comando em cada método, exceto quando o método retorna algum objeto, porque existe a exigência de que o retorno seja o último comando do método. É importante destacar que alguns métodos possuem muitos retornos, então nesses casos foi necessária uma mensagem de fim para cada retorno. Dessa forma, o código dos métodos se tornam muito poluídos.

Em contrapartida, para a implementação do rastreamento no ATM-AspectJ, foi necessária apenas a criação de um aspecto, apresentado na Figura 4.6. O aspecto denominado AspectTrace contém um *pointcut* e dois *advices*. O *pointcut* é responsável por interceptar todas as execuções de métodos incluindo construtores, *getters* e *setters*. Esse *pointcut* só não intercepta os métodos da própria classe AspectTrace.

```
1  package aspects;
2
3  import java.util.logging.Level;
10
11 @Aspect
12 public class AspectTrace {
13
14 private Logger logger= Logger.getLogger("trace"); //obtain the logger object
15
16     AspectTrace(){
17         logger.setLevel(Level.ALL); //initializing the log level
18     }
19
20     @Before("execution(* *.*(..)) || execution (*.new(..)) && " +
21             "!within(AspectTrace)&& !within(AspectLog)&& !within(AspectEH)")
22     public void beforeTrace(JoinPoint.StaticPart thisJoinPointStaticPart){
23         if(logger.isLoggable(Level.INFO)){
24             Signature sig = thisJoinPointStaticPart.getSignature();
25             logger.logp(Level.INFO, sig.getDeclaringType().getName(), sig.getName(), "begin ");
26         }
27     }
28
29     @After("execution(* *.*(..)) || execution (*.new(..)) && " +
30             "!within(AspectTrace)&& !within(AspectLog)&& !within(AspectEH)")
31     public void afterTrace(JoinPoint.StaticPart thisJoinPointStaticPart){
32         if(logger.isLoggable(Level.INFO)){
33             Signature sig = thisJoinPointStaticPart.getSignature();
34             logger.logp(Level.INFO, sig.getDeclaringType().getName(), sig.getName(), "end ");
35         }
36     }
37 }
```

Figura 4.6: Aspecto criado para fazer o rastreamento do ATM

O AspectTrace contém dois *advices*, um do tipo *before* e o outro *after*. O *advice* do tipo *before* executa antes da execução do *join point* interceptado pelo *pointcut*. Neste caso especificamente, o *advice* do tipo *before* escreve no *console* '*begin*' antes da execução de cada método do ATM-AspectJ. O *advice* do tipo *after* é responsável por escrever no *console* '*end* ' quando cada método terminar sua execução. Os dois *advices* escrevem, além de '*begin*' e '*end*', informações relativas ao método chamado como nome do método, da classe e pacote a qual ela pertece.

O código do aspecto responsável por fazer o *tracing* do software é apresentado na

Figura 4.6. É importante destacar que esse aspecto está pronto para ser reusado em qualquer sistema em Java que aceita AspectJ, pois não possui nenhum detalhe de implementação com os requisitos de negócio da aplicação.

Após a finalização da implementação do aspecto responsável pelo rastreamento do ATM foi realizada a fase de testes para avaliar se o rastreamento produz o mesmo resultado para as versões POO e POA do ATM. As duas versões foram executadas com as mesmas tarefas, e as saídas produzidas no *console* foram analisadas usando o programa WinMerge [Win] que compara linha a linha dois arquivos textos.

4.3.3 Implementação do tratamento de exceção

Para a implementação do tratamento de exceção como aspecto, primeiramente, uma busca com a palavra '*catch*' foi feita no ATM com objetivo de localizar todos os lugares onde são realizados tratamento de exceções. Foram listados 24 pontos no sistema onde o bloco *try-catch* é utilizado.

O primeiro problema encontrado foi que o tratamento de exceção do ATM é muito ineficiente. A maioria dos blocos *try-catch* não estavam implementados efetivamente para tratar a exceção. Para exemplificar, o código da Figura 4.7 mostra o método display da classe SimDisplay do ATM que possui um bloco *try-catch* que captura uma exceção do tipo Exception, contudo não realiza qualquer procedimento quando a exceção é lançada, como pode ser observado na linha 77.

```
62    /** Add text to the display - may contain one or more lines delimited by \n
63     *  @param text the text to display
64     */
65    void display(String text)
66    {
67        logger.log(Level.INFO, "begin " );
68
69        StringTokenizer tokenizer = new StringTokenizer(text, "\n", false);
70        while (tokenizer.hasMoreTokens())
71        {
72            try
73            {
74                displayLine[currentDisplayLine ++].setText(tokenizer.nextToken());
75            }
76            catch (Exception e)
77            { }
78        }
79
80        logger.log(Level.INFO, "end " );
81
82    }
```

Figura 4.7: Exemplo de bloco try-catch na classe SimDisplay do ATM

O AspectJ fornece suporte específico para lidar com exceções. Existe uma palavra reservada *handler* que identifica pontos no código onde as exceções são capturadas. Para isso, os blocos *try-catch* devem continuar no código mesmo com a inserção dos aspectos, o que não produz a limpeza almejada no código. A Figura 4.8 contém um trecho do método readBills da classe BillsPanel da versão AspectJ do ATM. A chamada de método dentro

do escopo do *catch* está comentada porque o responsável por realizar o tratamento da exceção na versão AspectJ do ATM é o advice do AspectEH (Figura 4.9).

```
83          try
84          {
85              billsNumber = Integer.parseInt(billsNumberField.getText());
86              if (billsNumber >= 0)
87                  validNumberRead = true;
88              else
89                  painel.getToolkit().beep();
90          }
91          catch(NumberFormatException e)
92          {
93              //a.getToolkit().beep();
94          }
```

Figura 4.8: Exemplo de bloco try-catch na classe BillsPanel do ATM-AspectJ

Um exemplo de advice do AspectEH é apresentado na Figura 4.9. Esse advice está anotado com @Before, significando que será executado assim que o compilador verificar o estouro de uma exceção do tipo NumberFormatException no método readBills da classe BillsPanel. As linhas 68-69 da Figura 4.7 substituem o código comentado (linha 93) da Figura 4.8.

```
66  @Before("withincode(* simulation.BillsPanel.readBills(..)) && handler(NumberFormatException)")
67  public void eh12(JoinPoint jp){
68      BillsPanel bp=(BillsPanel)jp.getThis();
69      bp.getPainel().getToolkit().beep();
70  }
```

Figura 4.9: Exemplo de advice do AspectEH para tratamento de exceção

4.4 Extração das métricas do ATM

Nesta seção a etapa de coletar métricas do ATM é relatada. Nas tabelas apresentadas nessa seção o ATM refatorado com aspectos é referido como ATM-AspectJ.

As ferramentas utilizadas para produzir as métricas a partir do código fonte foram o plugin Metrics for Eclipse e Sonar. Essas são ferramentas utilizadas pelos programadores para avaliar a qualidade do software produzido. Mais informações dessas ferramentas podem ser encontradas na Seção 2.3.

Na Seção 2.3 as métricas mais comumente utilizadas para avaliar a POA nos estudos de casos foram apresentadas. A proposta inicial era utilizar as mesmas métricas para avaliar o ATM, entretanto não foram encontradas ferramentas que abrangessem toda aquela gama de métricas. Existem poucas ferramentas para mensurar qualidade de software e algumas não são disponíveis gratuitamente.

O Plugin Metrics for Eclipse produz métricas relacionadas a tamanho e complexidade do código. Sendo assim, foi necessário a utilização de mais de uma ferramenta, pois a qualidade de software não pode ser decidida apenas com base no tamanho. O Sonar avalia outros fatores sobre a qualidade do software como qualidade do design e arquitetura e faz avaliações com relação a erros no código.

Nas subseções 4.4.1 e 4.4.2 são apresentados os resultados das métricas produzidos pelo plugin Metrics e pelo Sonar, respectivamente. Nas tabelas são apresentados os valores de métricas obtidos pelas duas versões do sistema, Java e AspectJ.

Todas as tabelas apresentadas nessa seção obedecem a uma padronização. O nome da métrica avaliada sempre é apresentado na primeira coluna da tabela. O resultado da métrica obtido pela versão original do software ATM pode ser observado na segunda coluna da tabela. Na terceira coluna pode-se observar o resultado da métrica obtido pela versão refatorada com AspectJ do ATM. A última coluna é reservada para a apresentação da avaliação da comparação dos resultados obtidos pelas duas versões do ATM. Se a célula da última coluna da tabela estiver preenchida com o símbolo +, significa que a versão AspectJ alcançou melhor resultado na métrica em comparação com a versão Java, representando que o resultado da métrica foi positivo para POA. Caso a célula da última coluna da tabela estiver preenchida com o símbolo -, o resultado da métrica foi favorável para a versão Java do ATM, revelando que o resultado da métrica foi negativo para POA. A célula da última coluna da tabela também pode estar preenchida com o símbolo ?, significando que os resultados da métrica foram iguais nas duas versões. Sendo assim, a avaliação é considerada inconclusiva, pois não foi possível estabelecer o melhor resultado para a métrica.

4.4.1 Avaliação com Plugin Metrics

Na Tabela 4.1, o resultado das métricas aplicadas no ATM e no ATM-AspectJ utilizando o plugin Metrics for Eclipse é apresentado.

Vale ressaltar que na Seção 4.2 o ATM foi apresentado com 2402 LOC e na Tabela 4.1 com 2941 LOC. Essa diferença significativa no número de linhas de código é porque foi implementado o interesse transversal rastreamento no ATM, com o objetivo de ter mais interesses transversais para se comparar. Portanto, efetivamente, este estudo de caso é da ordem de 3 KLOC.

Conforme pode ser observado, das 10 métricas apresentadas pelo Plugin Metrics, o ATM-AspectJ obteve 4 resultados ruins comparados com o ATM, 4 métricas não apresentaram diferença no resultado e apenas 2 métricas com piores resultados para o ATM.

4.4.2 Avaliação com Sonar

As tabelas dessa subseção são resultados das métricas do ATM e do ATM-AspectJ gerados pela ferramenta Sonar.

O Sonar apresenta uma grande lista de métricas para medir a qualidade do software. Algumas métricas não possuem descrição detalhada na documentação do Sonar. Dessa forma, os critérios utilizados para produzir um determinado resultado são desconhecidos. Por exemplo, na Tabela 4.5 o Sonar apresentou que o ATM possui 94,4% e o ATM-AspectJ

Métrica	**ATM**	**ATM AspectJ**	**Avaliação**
Linhas de código	2941	2575	+
Número de classes	42	45	-
Número de métodos	166	194	-
Número de atributos	89	88	+
Métodos ponderados por classe	313	354	-
Coesão nos métodos [Max]	0.869	0.906	-
Acoplamento Eferente [Max]	8	8	?
McCabe Complexidade Ciclomática [Max]	16	16	?
Número de filhos	11	11	?
Árvore de profundidade de Herança [Max]	4	4	?

Tabela 4.1: Comparação Métricas ATM e ATM refatorado

92,3% de usabilidade. Contudo, não se sabe em quais dados a ferramenta se baseou para gerar esse resultado.

As tabelas foram separadas em categorias de acordo com a propriedade de cada métrica:

- Tamanho - Tabela 4.2
- Complexidade - Tabela 4.3
- *Design* - Tabela 4.4
- *Rules Categories* - Tabela 4.5
- Regras - Tabela 4.6
- Geral - Tabela 4.7

Tamanho			
Métrica	**ATM**	**ATM AspectJ**	**Avaliação**
Linhas físicas	5755	5010	+
Linhas de código	2941	2575	+
Pacotes	6	7	-
Classes	42	45	-
Arquivos	37	40	-
Acessores	0	34	-
Métodos	188	191	-
API pública	165	161	-
Comandos	1350	993	+

Tabela 4.2: Comparação Métricas de Tamanho ATM e ATM refatorado geradas pelo Sonar

As métricas do Sonar que fornecem uma idéia do tamanho do sistema foram agrupadas na Tabela 4.2. A versão POA do ATM apresentou melhores resultados de métricas com relação ao número de linhas de código e números de comandos. Com base nesses

resultados, pode-se afirmar que POA utiliza menos comandos e linhas para desempenhar as mesmas funcionalidades apresentadas na versão POO. Em contrapartida, na versão POA o número de pacotes, classes, arquivos, métodos, acessores (número de *getters* e *setters*) e API pública (número de classes, métodos e propriedas públicas) aumentou, demonstrando que POA inclui novas estruturas ao código. Número maior de pacotes, classes, arquivos, métodos, acessores e API pública é considerado um resultado negativo porque POA precisou utilizar um maior número de estruturas enquanto POO necessita de menos estruturas para desempenhar as mesmas funcionalidades.

As métricas do Sonar que calculam a complexidade do sistema estão dispostas na Tabela 4.3. O Sonar disponibiliza os valores para complexidade baseado no cálculo da complexidade ciclomática, também conhecida como métrica McCabe [McCabe 1976]. A complexidade geral e a complexidade média dos métodos são maiores para a versão POA do ATM. Entretanto, a complexidade média de classes e arquivos são piores na versão POO do ATM. Os resultados das métricas de complexidade apresentados para o ATM e o ATM refatorado estão divididos. Além disso, os valores são bastante próximos provavelmente devido ao tamanho pequeno do sistema.

Complexidade			
Métrica	**ATM**	**ATM AspectJ**	**Avaliação**
Complexidade	386	392	-
Complexidade média dos métodos	2	2,1	-
Complexidade média das classes	9,2	8,7	+
Complexidade média dos arquivos	10,4	9,8	+

Tabela 4.3: Comparação Métricas ATM e ATM refatorado

O Sonar também disponibiliza métricas que avaliam a qualidade do *Design* do software. Os resultados das métricas sobre o *Design* das duas versões do ATM são apresentados na Tabela 4.4.

Design			
Métrica	**ATM**	**ATM AspectJ**	**Avaliação**
Resposta para classe	20	15	+
Falta de coesão de métodos	1	1,9	-
Índice de pacotes entrelaçados	12%	0%	+
Acoplamento entre objetos	71,9%	77,3%	-
Qualidade do Design	91%	87,6%	-
Profundidade de herança	100%	100%	?
Complexidade de classes e métodos	86,5%	88,1%	-

Tabela 4.4: Comparação Métricas ATM e ATM refatorado

As duas versões do ATM obtiveram o mesmo valor para a métrica Profundidade de herança. Isso significa que POA não provoca aumento nos níveis de herança dos objetos.

A métrica Resposta para classe mede a complexidade da classe com relação a chamada de métodos. Baseado nessa métrica, pode-se afirmar que o ATM versão POO faz mais chamadas a outros métodos, consequentemente o ATM refatorado consegue diminuir esse valor.

Outro resultado que foi melhorado com a inserção dos aspectos foi o índice de pacotes entrelaçados. Esse índice é calculado com base no nível de dependência cíclica entre os arquivos de diferentes pacotes, ou seja, o arquivo **X** do pacote **b** depende do arquivo **Y** do pacote **a** e vice e versa. O melhor valor é 0 % que significa que não há ciclos. A versão POO obteve 12% de índice de pacotes entrelaçados e a versão POA 0%. Isso significa que nesse caso POA alcançou o valor ideal e de alguma forma conseguiu eliminar os ciclos de dependência entre os pacotes.

A versão POA do ATM alcançou piores resultados nas métricas Falta de coesão de métodos, acoplamento entre objetos, qualidade do design e complexidade de classes e métodos. Com a melhora na separação dos interesses, era esperado que a versão POA melhorasse os resultados das métricas que medem a coesão. Contudo, não foi o que ocorreu e a possível causa é apresentada a seguir.

Para medir a coesão das classes do software o Sonar utiliza a métrica LCOM4 (Lack of Cohesion of Methods) baseada em Hitz & Montazer . LCOM4 é uma métrica para sistemas POO que calcula o número de componentes conectados na classe. Um componente conectado é um conjunto de métodos relacionados. Dois métodos **a** e **b** são relacionados se eles acessam uma mesma váriavel ou **a** chama **b**, ou **b** chama **a**. LCOM4 calcula de forma correta o nível de coesão de programas POO, porém não calcula coesão de forma justa para programas POA, já que um advice de um aspecto nunca chama outro advice e muito dificilmente acessam uma mesma variável. Portanto, os resultados da métrica de coesão claramente irão favorecer a versão POO, o que não significa necessariamente que a coesão é melhor com POO.

A Tabela 4.5 apresenta resultados de cinco categorias: Eficiência, Manutenabilidade, Portabilidade, Confiabilidade e Usabilidade. O Sonar não disponibiliza descrição detalhada sobre como são calculados esses valores.

Rules categories			
Métrica	**ATM**	**ATM AspectJ**	**Avaliação**
Eficiência	99,8%	99,8%	?
Manutenabilidade	94,2%	97,1%	+
Portabilidade	100%	100%	?
Confiabilidade	95,2%	93,4%	-
Usabilidade	94,4%	92,3%	-

Tabela 4.5: Comparação Métricas ATM e ATM refatorado

Outra funcionalidade importante fornecida pelo Sonar é a detecção de erros. Erros de compilação e execução, e alertas são contabilizados e apontados na Tabela 4.6.

Regras			
Métrica	**ATM**	**ATM AspectJ**	**Avaliação**
Total violações	279	232	+
violações pequenas	176	142	+
violações grandes	101	83	+
violações de informações	2	7	-
violações críticas	0	0	?
violações bloqueantes	0	0	?
índice de regras cumpridas	83,7%	84,9%	+
qualidade do código	89,5%	87,2%	-

Tabela 4.6: Comparação Métricas ATM e ATM refatorado

O cálculo da Tabela 4.7 apresenta a qualidade total do software. Os valores da duas versões do ATM são muito próximos, uma diferença de 1,5% a mais para a versão POO do ATM.

Geral			
Métrica	**ATM**	**ATM AspectJ**	**Avaliação**
qualidade total	67,1%	65,6%	-

Tabela 4.7: Comparação Métricas ATM e ATM refatorado

4.5 Discussão dos resultados

O estudo de caso proposto nesse capítulo tem por objetivo avaliar os benefícios/malefícios que a reengenharia do código fonte utilizando programação orientada a aspectos proporciona. A avaliação do software foi feita com base nas métricas de qualidade de software. Na Seção 4.4 todos os resultados das métricas geradas pelo plugin Metrics for Eclipse e Sonar foram apresentados.

Nesse caso, a POA utiliza menor número de linhas em comparação com POO. Esse resultado contraria a revisão sistemática, apresentada no capítulo 3. Nela, POA obteve resultado negativo em 7 artigos (que somam 15 aplicações - Tabela 3.3 e 3.5) para a métrica que mede linhas de código, ou seja, para a maioria dos estudos de caso encontrados temos que POA aumenta o número de linhas no código. O motivo da versão POA do ATM possuir menor número de linhas é atribuído a implementação do rastreamento como aspecto. Como as duas versões do ATM são comparadas, pode-se perceber que a versão POO utiliza um número de linhas muito maior para implementar rastreamento com Java.

Com relação às métricas relativas ao tamanho de código, com exceção das métricas que contabilizam o número de comandos e o número de linhas, todas as outras métricas (número de classes, métodos, arquivos, atributos) foram negativas para POA. Considerando que a proposta de POA é melhorar a separação de interesses, esses resultados

não são motivos para inviabilizar o uso de POA, porque a inserção de novas unidades podem trazer benefícios com relação à melhora na modularidade, manutenabilidade e compreensão do código.

As duas ferramentas, Metrics e Sonar, apesar de apresentarem valores de forma diferente, afirmaram que a coesão é melhor na versão POO do ATM. Esse resultado é surpreendente porque como foi apresentado na Seção 4.3, o código fica visualmente mais coeso na versão POA. A explicação para tal resultado pode ser o fato de que a métrica LCOM foi criada para medir coesão em softwares escritos em POO, e portanto favorece POO em detrimento a POA.

Sobre a complexidade apresentada pelas duas versões do ATM, não foi possível estabelecer qual é o paradigma mais adequado para a diminuição de complexidade, porque os valores são muito próximos e não são unânimes.

Os resultados das métricas produzidas pelo Sonar que avaliam o Design do ATM mostram que a diferença de qualidade é muito pequena, mas é favorável para a versão POO do ATM. Os resultados das métricas que contabilizam o número de erros do ATM são favoráveis para a versão POA do ATM. Pode-se concluir que, nesse caso, POA solucionou problemas e violações apresentadas na versão POO.

Baseado nas experiências adquiridas durante a fase de refatoração do ATM, análises qualitativas são apresentadas a seguir para discutir e indicar o melhor paradigma para cada tipo de interesse transversal.

A implementação do interesse transversal *logging* foi benéfica em POA no sentido de melhorar a coesão do código que implementa interesses principais. Outro ponto vantajoso na utilização de POA para implementação de *logging* é que toda implementação de *logging* está separada em uma unidade e não espalhada em diversas partes do código. Essa estratégia facilita a manutenção do software pois a alteração de alguma informação pode ser feita em apenas um módulo do software e não em diversos como na versão POO. Além disso, a inserção de uma regra para *logging* faz-se necessária apenas com a adição de um novo comando quantificado.

Com relação a implementação de Tratamento de Exceção nesse estudo de caso, foi possível perceber que não é muito indicado implementar Tratamento de exceção em softwares pequenos. Na versão original do ATM, o tratamento de exceção foi implementado de forma ineficiente, e asssim, muitos blocos *try-catch* não realizavam o tratamento adequado quando uma exceção ocorria. Além disso, como foi pontuado anteriormente, o AspectJ fornece um suporte específico para modularizar tratamento de exceção usando o *handler*. Tal suporte força a permanência das palavras *try-catch* no código que implementa os interesses transversais, o que não proporciona a limpeza total almejada no código principal. Portanto, criar novas estruturas como aspectos e *advices* para substituir poucas linhas pode não ser vantajoso para tratar exceção.

A implementação do interesse transversal rastreamento com POA foi positiva. O

esforço de implementar rastreamento com Java não se compara com a facilidade de implementação utilizando AspectJ. Enquanto que para a versão Java do ATM foi necessário alterar todos os métodos do software, na versão AspectJ foi necessário apenas a criação de dois *advices*. A implementação de rastreamento com POA tornou o código visivelmente mais coeso, melhorou o nível de reuso, facilitou a manutenção e diminuiu muito o número de linhas de código em comparação com a versão Java.

Capítulo 5

Estudo de Caso 2: JMoney

A proposta do segundo estudo de caso é avaliar a refatoração de software com aspectos em softwares de tamanho maior que 5 mil linhas de código. A partir da comparação dos resultados das métricas produzidas pelo software com versões POO e POA serão realizadas análises para determinar qual paradigma fornece mais benefícios. A proposta de desenvolver dois estudos de caso possibilita a formulação de conclusões mais consistentes, devido à quantidade maior de resultados. Portanto, as análises finais também serão baseadas na comparação dos dois estudos de caso.

O software JMoney, utilizado como segundo estudo de caso, foi escolhido a partir de um repositório de código fonte considerando critérios previamente estabelecidos. O processo de escolha é detalhado neste capítulo.

JMoney é um software de gerenciamento financeiro pessoal. A versão orginal, disponível no site oficial, possui mais de 8 KLOC. No estudo de caso foi utilizada uma versão alterada da ordem de 10 KLOC.

5.1 Critérios para escolha do software

Os critérios utilizados para o estudo de caso ATM também serviram de diretrizes para a escolha do segundo estudo de caso:

- Software com código fonte livre para download;
- Software codificado na linguagem orientada a objetos Java;
- Software que possui interesses transversais para serem transformados em aspectos;
- Software com tamanho maior que 5 KLOC;
- Software com código fonte compilando e executando corretamente.

O software JMoney foi escolhido no repositório de código fonte Qualitas Corpus [Qua]. Qualitas Corpus é um sistema que armazena diversas versões de 111 softwares de código

aberto escritos em Java. Dessa forma, os dois primeiros critérios para escolha já foram atendidos.

No Qualitas Corpus os softwares são separados em categorias definidas de acordo com o domínio da aplicação. Duas categorias de sistemas foram escolhidas, ferramentas e *middleware*. Estas categorias foram escolhidas porque seus sistemas possuem interesses transversais implementados de forma espalhada e entrelaçada.

Outro filtro utilizado para diminuir a lista de sistemas fornecida pelo Qualitas Corpus foi o tamanho. Somente softwares de tamanho menor ou igual a 20 KLOC foram considerados. Dessa forma, o número de opções diminuiu de 111 para 10. A lista com 10 softwares foi ordenada em ordem crescente de tamanho. O primeiro sistema da lista denominado *oscache* está indicado como *dead* pelo Qualitas Corpus. O *oscache* não atendeu todos os critérios, porque seu código fonte não está mais disponível para copiar a partir do site. Por isso, o segundo sistema, JMoney, foi escolhido devido ao fato de que ele satisfez todos os critérios.

5.2 Apresentação do Software

JMoney [JMo] é um software de gerenciamento financeiro pessoal codificado em Java. JMoney possui licença pública geral GNU. Este sistema suporta múltiplas contas em diferentes moedas, entradas de dois bancos, categorias de receitas e despesas, vários relatórios e troca de arquivos Quicken (QIF).

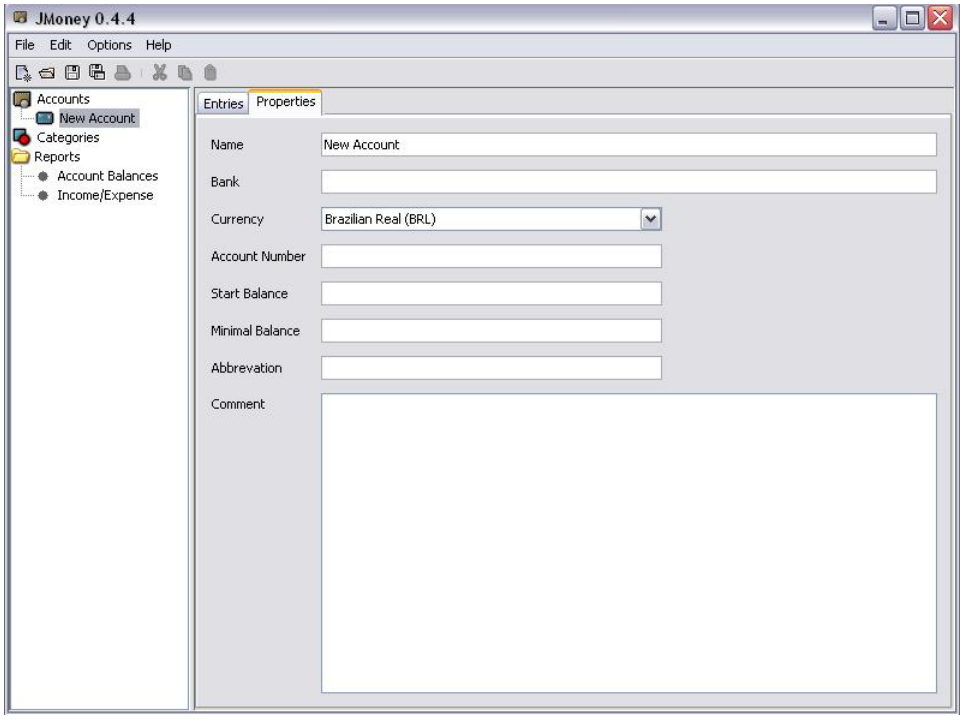

Figura 5.1: Tela do JMoney

A tela principal do JMoney com a criação de uma nova conta pode ser observada na Figura 5.1.

Diferentemente do primeiro estudo de caso, o JMoney não possui ampla documentação disponível. Portanto, não há acesso no site para diagramas UML e descrição detalhada de suas funcionalidades. A documentação disponível do JMoney se limita a pouco mais de um parágrafo de descrição. Isso dificultou o processo de identificação dos interesses transversais, além de tornar a compreensão do código para refatoração mais demorada.

5.3 Refatoração do JMoney

O processo de refatoração do JMoney foi desenvolvimento de forma similar ao do ATM. Os interesses transversais foram identificados, em seguida foi iniciada a etapa de implementação dos aspectos, posteriormente a fase de testes, finalmente a aplicação das métricas e a comparação.

Três tipos de interesses transversais foram implementados como aspectos no JMoney: rastreamento, tratamento de exceção e checagem de pontos nulos. Como ocorrido com o estudo de caso ATM, JMoney não possuia rastreamento implementado no código fonte original. Portanto, esse interesse transversal foi implementado na versão Java do sistema utilizando apenas os recursos fornecidos pela POO. Assim, o código fonte original foi modificado durante as atividades da pesquisa e as linhas de código aumentaram. O JMoney passou a possuir em torno de 10 KLOC.

5.3.1 Implementação do Rastreamento

A fase de implementação foi iniciada com a implementação do interesse transversal rastreamento em Java na versão original do JMoney. A codificação de rastreamento em Java exige grande esforço do programador, porque requer a alteração de todos os métodos do sistema. Incluir mensagens que sinalizam o início e o fim de 568 métodos, como é o caso do JMoney, é muito trabalhoso.

A tarefa de implementar rastreamento com AspectJ na versão POA do JMoney foi mais simples do que na versão POO. O aspecto criado para realizar o rastreamento no primeiro estudo de caso ATM foi reutilizado. Apenas duas linhas do aspectTrace do ATM foram modificadas para que o rastreamento funcionasse no JMoney. Conforme pode ser observado na Figura 5.2, o aspectTrace do ATM é idêntico ao aspectTrace do JMoney.

Os *advices beforeTrace* e *afterTrace* possuem o mesmo pointcut (linhas 20-21 e 29-30 do código apresentado na Figura 5.2). O poincut define os pontos que serão interceptados pelo aspecto durante a execução do sistema. Para realizar rastreamento, todos os métodos foram interceptados, inclusive métodos construtores. Para interceptar todos os métodos foi utilizada a expressão *execution(* *.*(..))*, que significa que todos os métodos devem

```
1  package net.sf.jmoney.aspectos;
2
3  import java.util.logging.Level;
10
11 @Aspect
12 public class AspectTrace {
13
14 private Logger logger= Logger.getLogger("trace"); //obtain the logger object
15
16     AspectTrace(){
17         logger.setLevel(Level.ALL); //initializing the log level
18     }
19
20     @Before("execution(* *.*(..)) || execution (*.new(..)) && !within(AspectTrace) && " +
21             "!within(AspectEH) && !within(AspectNPC)")
22     public void beforeTrace(JoinPoint.StaticPart thisJoinPointStaticPart){
23         if(logger.isLoggable(Level.INFO)){
24             Signature sig = thisJoinPointStaticPart.getSignature();
25             logger.logp(Level.INFO, sig.getDeclaringType().getName(), sig.getName(), "begin ");
26         }
27     }
28
29     @After("execution(* *.*(..)) || execution (*.new(..)) && !within(AspectTrace) && " +
30             "!within(AspectNPC)&& !within(AspectEH)")
31     public void afterTrace(JoinPoint.StaticPart thisJoinPointStaticPart){
32         if(logger.isLoggable(Level.INFO)){
33             Signature sig = thisJoinPointStaticPart.getSignature();
34             logger.logp(Level.INFO, sig.getDeclaringType().getName(), sig.getName(), "end ");
35         }
36     }
```

Figura 5.2: Aspecto reutilizado do ATM para fazer o rastreamento do JMoney

ser interceptados independemente dos modificadores de acesso (*private*, *protected*, *public* ou *static*), da classe, da quantidade e dos tipos de parâmetros. Todos os construtores foram interceptados utilizando a expressão *execution(*.new(..))*.

A parte final da expressão que define o pointcut (*!within(AspectTrace) !within(AspectEH) !within(AspectNP)*) estabelece que os métodos das classes AspectTrace, AspectEH e AspectNP não devem ser interceptados. Os rastros de execução produzidos pelos aspectos da versão POA não foram considerados visando facilitar a comparação dos rastros de execução das duas versões do JMoney na fase de testes.

Os testes foram realizados para comparar se a implementação do rastreamento com aspectos não alterou as funcionalidades que são apresentadas na versão original do JMoney. Os rastros produzidos pelas duas versões do JMoney foram comparados usando uma ferramenta chamada WinMerge [Win]. O WinMerge compara dois arquivos e apresenta visualmente as diferenças textuais entre eles. Os rastros de execução produzidos durante a execução de uma mesma tarefa nas duas versões do JMoney foram idênticos, comprovando que as funcionalidades originais foram preservadas.

5.3.2 Implementação do Tratamento de Exceção

O início da implementação do tratamento de exceção com aspectos foi marcado pela localização dos pontos no código que tratam as exceções. No código do JMoney foram localizados 50 lugares onde é feito tratamento de exceção utilizando blocos *try-catch*.

Para cada bloco *try-catch* foram criados pointcut e advice, com objetivo de tratar a

exceção em uma unidade separada denominada AspectEH.

O AspectJ fornece suporte específico para lidar com exceções. O *handler* identifica pontos no código onde as exceções são capturadas. Para isso, os blocos try-catch devem continuar no código mesmo com a inserção dos aspectos, o que não produz a limpeza almejada no código.

O código do método *updateMinBalance* da classe *AccountPropertiesPanel* da versão AspectJ do JMoney é mostrado na Figura 5.3. Conforme pode ser observado na linha 305, uma seta sinaliza a existência de um pointcut que intercepta esse *joinpoint*. Esse *joinpoint* é um *catch* que captura exceções do tipo *ParseException* e foi definido no pointcut do AspectEH (linha 100 da Figura 5.4). A chamada de método dentro do escopo do *catch* está comentada (linha 306 da Figura 5.3) porque o responsável por realizar o tratamento da exceção na versão AspectJ do JMoney é o advice do AspectEH (Figura 5.4). As linhas 102-103 substituem a antiga implementação do tratamento de exceção do método *updateMinBalance*.

```
301    void updateMinBalance() {
302        try {
303            NumberFormat.getNumberInstance().parse(minBalanceField.getText());
304            minBalance = new Long(account.parseAmount(minBalanceField.getText()));
305        } catch (ParseException pex) {
306            // minBalance = null;
307        }
308        setBalanceFields();
309    }
```

Figura 5.3: Exemplo de bloco *try-catch* na classe *AccountPropertiesPanel* do JMoney-AspectJ

```
 99    @Before("handler(ParseException) && " +
100            "withincode(* net.sf.jmoney.gui.AccountPropertiesPanel.updateMinBalance(..))")
101    public void exceptionHandler5(JoinPoint jp) {
102        AccountPropertiesPanel a =(AccountPropertiesPanel) jp.getThis();
103        a.setMinBalance(null);
104    }
```

Figura 5.4: Exemplo de *advice* do AspectEH para tratar exceção de método na classe *AccountPropertiesPanel* do JMoney-AspectJ

Algumas restrições do AspectJ tornaram-se obstáculos durante a fase de implementação. O acesso ao contexto do joinpoint pelo aspecto não inclui acesso à variável local, ou seja, acesso à variável declarada e utilizada apenas em um método. Portanto, em situações onde era necessária a utilização de uma variável local, foi necessário transformá-la em variável de classe. Essa solução não representa uma boa alternativa, porque conceitualmente, variáveis de classe devem ser atributos que representam um determinado objeto. Contudo, essa foi a solução encontrada para reproduzir os comandos originais.

Outra restrição do AspectJ é com relação à utilização do plugin AspectJ no Eclipse. Não há atalhos para fácil acesso a classes dentro do aspecto. Por exemplo, o programador em Java ao declarar um objeto pode ter acesso à implementação da classe apertando a tecla *control* e com um *click* tem acesso à implementação da classe. Esse atalho não existe

na declaração de um *pointcut* no aspecto. Disponibilizar esse atalho poderia garantir mais agilidade ao programador durante o processo de desenvolvimento de aplicações.

A compreensão limitada do sistema, devido ao pouco contato com a documentação, tornou mais complicada a tarefa de testes. Para cada caso do bloco *try-catch* o código era depurado para verificar em quais situações o sistema executava o catch. Posteriormente, o mesmo era feito na nova versão POA para verificar se também executava o catch.

5.3.3 Implementação da Checagem de Pontos Nulos

Para realizar a implementação do interesse transversal Checagem de Pontos Nulos como aspecto, primeiramente foi necessário localizar no código do JMoney os lugares onde essas checagens acontecem. Utilizando o Eclipse, foi feita uma busca no código para localizar todas as ocorrências da palavra *null*. A palavra reservada *null* pode ser usada em diversas situações, como por exemplo, atribuições, parâmetro nas chamadas de funções e até mesmo em comentários. Todos esses casos foram listados o que gerou uma grande lista de 264 itens.

Os casos em que *null* é utilizado como parte de um comentário e como atribuição foram retirados da lista, restando assim, 150 ocorrências onde é feito o teste condicional utilizando IF. A lista diminui muito quando os IF's que não são relevantes são excluídos porque o teste condicional foi utilizado apenas para direcionar o fluxo de execução. Os IF's relevantes são aqueles que verificam se um determinado objeto é nulo antes de acessá-lo através de atributos e métodos.

O código do método updateUI da classe CategoryPanel apresentado na Figura 5.5 exemplifica um caso de um IF considerado relevante porque realiza Checagem de Ponto Nulo. Na linha 71 acontece uma checagem de ponto nulo, utilizando um IF, com o objetivo de evitar uma exceção do tipo *NullPointerException* caso o objeto seja nulo.

```
69    public void updateUI() {
70        super.updateUI();
71        if (categoryTree != null)
72            categoryTree.setCellRenderer(new CategoryTreeCellRenderer());
73    }
```

Figura 5.5: Exemplo de checagem de pontos nulos da classe CategoryPanel do JMoney

Algumas ocorrências onde acontece a checagem de pontos nulos foram tratadas e implementadas como aspecto. O código em aspecto responsável por realizar a checagem de pontos nulos do método updateUI é apresentado na Figura 5.6. Foram necessários dois advices para desempenhar a tarefa de checagem de pontos nulos. O *advice* anotado com @Before é responsável por interceptar o fluxo do programa toda vez que a variável categoryTree for acessada. Nesse método, seguindo a lógica do código, se a variável categoryTree for nula, o sistema não pode executar os comandos dentro do escopo do IF e deve executar a próxima instrução depois do IF. O advice do tipo *before* não tem permissão

para mudar o fluxo de execução de um método, apenas o *advice* do tipo *around* tem essa característica. Portanto, nas linhas 24-32 o *advice* do tipo *around* realiza a checagem de pontos nulos e faz a tomada de decisões de acordo com o resultado da checagem. A linha 26 é responsável por acessar o contexto do *JoinPoint*. Todos os atributos e métodos da classe *CategoryPanel* podem ser acessados a partir desse instante. Na linha 27, a checagem de pontos nulos é realizada para verificar o atual estado da variável *categoryTree*. Se essa variável não for nula, o *advice* usa o comando *proceed* para determinar que o método *updateUI* continue sendo executado normalmente. Caso a variável seja nula o *advice* determina que o método *updateUI* seja finalizado imediatamente, preservando, assim, a lógica original do método *updateUI*.

Aparentemente, o *advice* do tipo *before* não produz nenhum efeito nesse processo porque não tem comandos no seu escopo. Contudo, ele desempenha um papel importante que é demarcar o início da atuação do *advice* do tipo *around*. Se o *advice* do tipo *before* não existisse, o *advice* do tipo *around* faria a checagem de pontos nulos para a variável. Caso a variável fosse diferente de *null*, então executaria a chamada do *proceed*, ou caso contrário *return*. Contudo, os comandos anteriores à checagem de pontos nulos não seriam executados no segundo caso, o que não corresponderia à lógica original dos métodos.

```
18    //Inicio=Before e Around para método updateUI
19    @Before("withincode(* net.sf.jmoney.gui.CategoryPanel.updateUI(..)) " +
20            "&& get(javax.swing.JTree categoryTree) ")
21    public void InitNullPointerCheck1(JoinPoint jp){
22    }
23
24    @Around("execution(* net.sf.jmoney.gui.CategoryPanel.updateUI(..))")
25    public void NullPointerCheck1(JoinPoint jp, ProceedingJoinPoint pjp) throws Throwable{
26        CategoryPanel a=(CategoryPanel) jp.getThis();
27        if(a.getCategoryTree() != null){
28            pjp.proceed();
29        }else{
30            return;
31        }
32    }
33    //Fim=Before e Around para método updateUI
```

Figura 5.6: Exemplo de *advice* para checagem de pontos nulos do método updateUI da classe CategoryPanel do JMoney

Após a inserção do aspecto que realiza a checagem de pontos nulos, o método updateUI da classe CategoyPanel sofreu alterações. Não há mais o teste condicional para testar se a variável categoryTree é nula. Os advices da Figura 5.6 executam a checagem e impedem a execução da linha 71 da Figura 5.7 caso fique constatado que a variável é nula.

```
69    public void updateUI() {
70        super.updateUI();
71        categoryTree.setCellRenderer(new CategoryTreeCellRenderer());
72    }
```

Figura 5.7: Exemplo de checagem de pontos nulos da classe CategoryPanel do JMoney

Para cada checagem de pontos nulos implementada como aspectos foram realizados testes. Os testes objetivam garantir que o novo JMoney, refatorado com POA, preserve as mesmas funcionalidades do JMoney original.

5.4 Extração das métricas do JMoney

Finalizada a etapa de implementação, as métricas da versão original e da versão refatorada do JMoney foram extraídas com o auxílio das ferramentas Metrics e Sonar. Os resultados coletados das métricas estão apresentados nas tabelas das subseções seguintes.

Para cada métrica apresentada nas tabelas, existe uma sinalição para demonstrar o desempenho da versão POA do JMoney em comparação com a versão POO. A última coluna de cada tabela pode estar preenchida com +, - ou **?**. O símbolo + sinaliza que a versão POA do JMoney conseguiu melhorar o valor da métrica em comparação com a versão POO, representando que o resultado da métrica foi positivo para POA. O símbolo - indica que a versão JMoney-AspectJ obteve pior resultado na métrica comparado com JMoney, sinalizando que o resultado da métrica foi negativo para POA. O ? significa que as duas versões do JMoney obtiveram resultados iguais para a métrica. Nesse caso, o resultado é considerado inconclusivo porque não foi possível avaliar se POA influencia de forma positiva ou negativa no valor da métrica.

5.4.1 Avaliação com Plugin Metrics

O Plugin do Eclipse Metrics atribuiu valores de diversas métricas para as duas versões do JMoney. Os resultados de 10 métricas são exibidos na Tabela 5.1.

Como pode ser observado na Tabela 5.1, a versão POA do JMoney diminui o número de linhas e melhorou a coesão do código, obtendo melhores resultados em comparação com a versão POO.

As duas versões do JMoney obtiveram o mesmo resultado para quatro métricas. Pode-se considerar um ponto positivo o fato de que POA não piora métricas com relação a acoplamento, complexidade ciclomática, número de filhos e profundidade de herança.

Como resultado negativo, em comparação com os resultados obtidos pela POO, POA apresentou piores resultados nas métricas que medem número de classes, atributos, métodos e métodos ponderados por classe. Os resultados dessas métricas são considerados piores para POA porque um número maior de classes, atributos e métodos pode produzir impactos negativos no código. Maior possibilidade de erros e maior esforço em testes unitários são exemplos de situações afetadas por um número grande de classes, atributos e métodos. A versão POO do sistema obteve resultados positivos nessas métricas porque conseguiu utilizar menos elementos para executar as mesmas funcionalidades que a versão POA.

5.4.2 Avaliação com Sonar

A ferramenta Sonar também disponibiliza resultados de métricas que determinam uma noção das dimensões do software. Essas métricas foram extraídas do JMoney e JMoney-

Métrica	JMoney	Jmoney AspectJ	Avaliação
Linhas de código	10052	8232	+
Número de classes	79	82	-
Número de métodos	568	622	-
Número de atributos	430	437	-
Métodos ponderados por classe	1101	1177	-
Coesão nos métodos [Max]	0.944	0.932	+
Acoplamento Eferente [Max]	21	21	?
McCabe Complexidade Ciclomática [Max]	33	33	?
Número de filhos	21	21	?
Árvore de profundidade de Herança [Max]	6	6	?

Tabela 5.1: Comparação Métricas JMoney e JMoney refatorado

AspectJ (Tabela 5.2). A maioria dos resultados da métricas de Tamanho favorecem a versão POO do JMoney. As métricas que calculam número de pacotes, classes, arquivos, acessores, métodos e API pública obtiveram piores resultados com a refatoração com POA. A versão POA conquistou melhores resultados nas métricas número de linhas de código e comandos.

O Sonar e o plugin Metrics apresentaram valores diferentes para o número de classes. Foi realizada uma checagem manual e o plugin Metrics apresenta o valor correto. Uma suposta razão para tal fato é que o Sonar contabiliza interfaces como classes. O JMoney possui 3 interfaces, o que provalvemente explica o fato de o Sonar contabilazar 82 classe para o JMoney enquanto o plugin Metrics apresentou 79.

Tamanho			
Métrica	**JMoney**	**JMoney AspectJ**	**Avaliação**
Linhas físicas	13346	11577	+
Linhas de código	10052	8232	+
Pacotes	4	5	-
Classes	82	85	-
Arquivos	54	57	-
Acessores	0	20	-
Métodos	713	747	-
API pública	577	610	-
Comandos	4904	3512	+

Tabela 5.2: Comparação Métricas de Tamanho JMoney e JMoney refatorado geradas pelo Sonar

Na Tabela 5.3, os resultados das métricas que medem a complexidade do código de cada versão do JMoney são expostos. Semelhantemente aos resultados apresentados para o ATM, os resultados sobre complexidade apresentados pelo Sonar são conflitantes. A complexidade geral é pior na versão POA. Contudo, a complexidade das classes e arquivos é pior para a versão POO. Outro ponto importante para análise é que a diferença entre os valores das métricas de complexidade é muito pequena. Em ambos estudos de caso,

POO e POA não se diferem com relação à complexidade. Pode-se concluir que a inserção de aspectos não produz grandes malefícios com relação à complexidade de código.

Complexidade			
Métrica	**JMoney**	**JMoney AspectJ**	**Avaliação**
Complexidade	1416	1464	-
Complexidade média dos métodos	2	2	?
Complexidade média das classes	17,3	17,2	+
Complexidade média dos arquivos	26,2	25,7	+

Tabela 5.3: Comparação Métricas JMoney e JMoney refatorado

Com exceção da métrica Resposta para Classe, todas as métricas que avaliam o Design das duas versões do JMoney definem a versão POO com maior qualidade de Design.

Design			
Métrica	**JMoney**	**JMoney AspectJ**	**Avaliação**
Resposta para classe	39	31	+
Falta de coesão de métodos	1	2,2	-
Índice de pacotes entrelaçados	10,9%	11,4%	-
Acoplamento entre objetos	83,5%	87,6%	-
Qualidade do Design	86,8%	81,1%	-
Profundidade de herança	97,6%	98,1%	-
Complexidade de classes e métodos	68,6%	68,7%	-

Tabela 5.4: Comparação Métricas JMoney e JMoney refatorado

Na Tabela 5.5, os valores das métricas da propriedade Rules Categories são apresentados. Embora a diferença seja notoriamente muito pequena, os resultados favorecem a versão POO do JMoney. Segundo os valores calculados pelo Sonar, a versão POO do JMoney possui melhores características de Eficiência, Manutenabilidade, Portabilidade, Confiabilidade e Usabilidade.

Rules categories			
Métrica	**JMoney**	**JMoney AspectJ**	**Avaliação**
Eficiência	99,7%	99,6%	-
Manutenabilidade	84,3%	83,3%	-
Portabilidade	99,3%	99,1%	-
Confiabilidade	91,9%	89,5%	-
Usabilidade	94,1%	90,8%	-

Tabela 5.5: Comparação Métricas JMoney e JMoney refatorado

As métricas relacionadas à detecção de erros foram agrupadas na Tabela 5.6. O Sonar aponta o número de erros de compilação, de execução, e alertas. Como pode ser observado, os números são muito parecidos. A versão POA do JMoney apresentou duas violações a mais comparada com a versão POO.

Regras			
Métrica	**JMoney**	**JMoney AspectJ**	**Avaliação**
Total violações	1634	1636	-
violações pequenas	839	838	+
violações grandes	750	758	-
violações de informações	45	40	+
violações críticas	0	0	?
violações bloqueantes	0	0	?
índice de regras cumpridas	69,3%	62,2%	-
qualidade do código	74,8%	71,3%	-

Tabela 5.6: Comparação Métricas JMoney e JMoney refatorado

O cálculo da Tabela 5.7 apresenta a qualidade total do software. Os valores das duas versões do JMoney são muito próximos, uma diferença de 2,4% a mais para a versão POO do JMoney.

Geral			
Métrica	**JMoney**	**JMoney AspectJ**	**Avaliação**
qualidade total	62,7%	60,3%	-

Tabela 5.7: Comparação Métricas JMoney e JMoney refatorado

5.5 Discussão dos resultados

O estudo de caso proposto nesse capítulo tem por objetivo avaliar as vantagens e desvantagens que a reengenharia do código fonte utilizando programação orientada a aspectos proporciona no software. A avaliação do software foi feita com base nas métricas de qualidade de software. Na Seção 5.4 todos os resultados das métricas geradas pelo plugin Metrics for Eclipse e Sonar foram apresentados.

Nesse caso, a POA utiliza menor número de linhas em comparação com POO. Esse resultado é muito interessante porque contraria a revisão sistemática, apresentada no capítulo 3. Nela, POA obteve resultado negativo para a métrica que mede linhas de código, ou seja, para a maioria dos estudos de caso encontrados tem-se que POA aumenta o número de linhas no código.

Com relação às métricas relativas ao tamanho de código, com exceção das métricas que contabilizam o número de comandos e o número de linhas, todas as outras métricas (número de classes, métodos, arquivos, atributos) foram negativas para POA. Considerando que a proposta de POA é melhorar a separação de interesses, esses resultados não são motivos para inviabilizar o uso de POA, porque a inserção de novas unidades podem trazer benefícios com relação à melhora na modularidade, manutenabilidade e compreensão do código.

Sobre a complexidade apresentada pelas duas versões do JMoney, não foi possível estabelecer qual é o paradigma mais adequado para a diminuição de complexidade, porque os valores são muito próximos e não são unânimes.

Os resultados das métricas produzidas pelo Sonar que avaliam o *Design* do JMoney mostram que a diferença de qualidade é muito pequena, mas é favorável para a versão POO do JMoney.

O Sonar disponibiliza resultados que avaliam a Eficiência, Manutenabilidade, Portabilidade, Confiabilidade e Usabilidade. Esses resultados são formulados com base em cálculos com diversas métricas. Nesse estudo, a versão POA obteve os piores resultados para todos esses atributos de qualidade.

Além dos resultados quantitativos obtidos pela extração das métricas, um resultado qualitativo importante foi encontrado. Pode-se observar claramente que a POA é extremamente benéfica para a implementação do interesse transversal rastreamento em comparação com POO. As vantagens de implementar rastreamento com aspectos são: menor número de linhas de código, menor esforço na implementação, reuso e manutenção. Os dois estudos de casos demonstraram que o número de linhas é menor com o uso de aspectos no software e o rastreamento é o principal responsável por essa melhora. O esforço para implementar rastreamento com POO é considerado exaustivo em comparação à facilidade de implementação com POA. O reuso da unidade de rastreamento com POA garante agilidade no processo de desenvolvimento de software. A manutenção do rastreamento com POO requer a alteração de todos os métodos do sistema, mas com POA a alteração é realizada apenas em um aspecto.

A implementação dos outros interesses transversais implementados com POA, *logging*, tratamento de exceção e checagem de pontos nulos, é benéfica com a relação à manutenção porque a alteração é realizada em apenas uma unidade, mas não proporcionam o nível de reuso apresentado pelo rastreamento. A unidade criada para fazer o rastreamento do software pode ser reusado em qualquer software sem grandes alterações, assim como foi reusada do ATM para o JMoney. Esse alto nível de reuso do rastreamento foi provocado pela declaração de *pointcuts* genéricos, isto é, os *pointcuts* foram definidos sem informações específicas de classes e métodos, conforme foi detalhado anteriormente. Diferentemente do rastreamento, as unidades que realizam *logging*, tratamento de exceção e checagem de pontos nulos são formuladas através de informações específicas de cada software, como nome de classe e métodos. Sendo assim, para reusar essas unidades são necessárias muitas alterações.

Capítulo 6

Conclusão e Trabalhos Futuros

Dois estudos de caso foram realizados para avaliar quantitativamente o impacto da POA no desenvolvimento de software. Dois softwares, ATM e JMoney, originalmente codificados em POO foram refatorados com POA para analisar qual paradigma produz mais benefícios em relação à qualidade de software. Interesses transversais que produziam código espalhado e entrelaçado na versão POO foram modularizados em unidades separadas na versão POA utilizando aspectos. A qualidade de software foi medida através de métricas geradas automaticamente por duas ferramentas, plugin Metrics e Sonar.

Os resultados obtidos nos estudos de caso mostraram que POA utiliza menor número de linhas de código em comparação com POO. Esse resultado contraria a revisão sistemática cujo resultado foi que na maioria dos estudos empíricos realizados, POA aumenta o número de linhas de código. A implementação do interesse transversal rastreamento foi considerada a principal responsável pela diminuição do número de linhas de código, porque a implementação em POA utiliza poucas linhas em relação à grande quantidade de linhas necessárias para a implementação em POO.

As métricas que contabilizam número de classes, pacotes, métodos, arquivos e atributos não foram favoráveis para POA. Nos dois estudos de caso, POA obteve resultados maiores em comparação com POO. Resultados ruins nessas métricas significam que POA utiliza mais estruturas e funções que POO para desempenhar as mesmas tarefas e por isso o paradigma POA pode se tornar mais complexo e suscetível a erros. Contudo, maior quantidade de classes, pacotes e métodos também pode indicar melhora na modularidade do sistema. Considerando que POA se propõe a melhorar a Separação de Interesses no software através dos aspectos, é natural o aumento no valor dessas métricas. Portanto, esses resultados não são motivos para inviabilizar o uso de POA, porque a inserção de novas unidades pode trazer benefícios com relação à melhora na modularidade, manutenabilidade e compreensão do código.

Sobre a complexidade nos softwares, os resultados das métricas não apresentaram evidências conclusivas. Nos dois estudos de casos as métricas que mediam complexidade geral eram negativas para POA, porém as métricas que mediam a complexidade de classes

e arquivos eram positivas para POA. Além disso, a diferença entre os resultados foi muito pequena. Baseado nesses resultados, pode-se afirmar que POA pode aumentar um pouco a complexidade do código, porém esse aumento pode ser considerado insignificante.

As métricas que medem o acoplamento no código obtiveram resultados inconclusivos no ATM e JMoney. Os resultados das métricas que avaliam o acoplamento, no geral, são iguais para as versões POO e POA. Sendo assim, não é possível concluir se o impacto de POA foi positivo ou negativo.

Os resultados das métricas que avaliam a coesão do software foram conflitantes. No estudo de caso ATM a coesão foi negativa para POA, revelando que POA piora a coesão do código em comparação com POO. Contudo, o estudo de caso JMoney apresentou um resultado contrário ao ATM. No segundo estudo de caso, POA melhorou a coesão. A melhora na Separação de Interesses proporcionada por POA deveria melhorar a coesão no código. Mas como foi pontuado no Capítulo 4, a forma com que as métricas calculam coesão favorecem o código escrito em POO.

As métricas que não foram favoráveis para POA possuem uma diferença pequena em relação a POO. Considerando que a maioria das métricas foram criadas para avaliar programas escritos em POO, os sistemas POA passaram pelo teste de qualidade. Dessa forma, é importante que as métricas geradas pelas ferramentas de extração automática de métricas sejam métricas que mensurem a qualidade do software independentemente do paradigma de programação utilizado no código fonte para tornar a comparação entre paradigmas mais justa.

A avaliação quantitativa baseada através de análises sobre os resultados das métricas de software possui como conclusão que em geral os resultados negativos para a versão POA são muito próximos da versão POO. Sendo assim, os resultados em geral não são suficientes para inviabilizar o uso de POA em refatorações.

Sobre a avaliação qualitativa, pode-se perceber que a refatoração com POA é mais indicada para interesses transversais que podem ser tratados de forma genérica como o rastreamento. Os demais interesses transversais considerados nos estudos de caso, *logging*, tratamento de exceção e checagem de pontos nulos, precisam ser tratados caso a caso, ou seja, cada *join point* interceptado recebe um tratamento específico. Essa situação promove o grande esforço para substituir poucas linhas no código principal pela criação de novas estruturas, como *pointcuts* e *advices*.

As questões de pesquisa propostas nesse trabalho foram respondidas a partir dos resultados obtidos nos estudos experimentais desenvolvidos.

1. **Qual o impacto da refatoração orientada a aspectos na qualidade do software?**

   Os estudos experimentais foram desenvolvidos com intuito de avaliar se a inserção de aspectos promove mais benefícios do que outros paradigmas existentes. A avaliação

de qualidade de software com POA foi medida a partir de métricas de software. Os resultados obtidos pelas versões POA dos dois softwares não foram todos favoráveis para POA. Contudo, pôde-se perceber que a diferença entre os resultados foram mínimas. Sendo assim, os resultados medidos pelas métricas não são suficientes para aconselhar contra o uso de POA. As vantagens e desvantagens de refatorar software com POA foram explicitadas durante todo o desenvolvimento desse trabalho. Resumidamente, os benefícios de refatorar código com POA listados incluem melhoria no reuso, manutenção e coesão de código devido à melhor separação de interesses. As desvantagens incluem esforço na inclusão de novas estruturas para separar poucas linhas de código no caso dos interesses tranversais tratamento de exceção, *logging* e checagem de pontos nulos.

2. **Como mensurar e avaliar a qualidade de POA em comparação com outros paradigmas como POO?**

   Um conjunto de métricas foi proposto para avaliar a qualidade de POA. Grande parte das métricas mais comumente utilizadas para avaliar qualidade de POA [França e Soares 2012] também foram utilizadas nesse estudo. Além disso, muitas outras métricas importantes que não foram consideradas em outros estudos foram avaliadas. Algumas métricas, como complexidade, número de linhas de código, classes, métodos e atributos são foram contabilizadas idependentemente do paradigma. Outras métricas, como coesão (LCOM), são originalmente propostas para avaliar especificamente o paradigma de programação orientada a objetos. Contudo, como os softwares refatorados com POA mantiveram a maior parte da implementação construída com POO (toda a lógica de negócios - interesses principais), a avaliação com métricas para POO também foi útil para avaliar POA.

Os resultados obtidos pelas métricas nos estudos empíricos encontrados na Revisão Sistemática foram reunidos e apresentados na terceira coluna da Tabela 6.1. Nas colunas quatro e cinco são apresentados os resultados das métricas obtidos pelo ATM e JMoney, respectivamente. Finalmente, na última coluna um sumário de todos os resultados é apresentado.

Propriedade	**Métrica**	**Rev. Sist.**	**ATM**	**JMoney**	**Total**
Tamanho do código	Linhas de código (LOC)	2+, 7-, 3?	+	+	4+, 7-, 3?
	Tamanho do Vocabulário	1+, 6-	-	-	1+, 8-
	Número de atributos	3+, 1?	+	-	4+, 1-, 1?
	Número de operações	1-	-	-	3-
	Peso das operações por componente	3+, 2-	-	-	3+, 4-
Complexidade	Complexidade Ciclomática por componente	1+	+	+	3+
Acoplamento	Acoplamento entre componentes	3+, 3-, 1?	-	-	3+, 5-, 1?
	Árvore de Profundidade de Herança	1+, 2-	?	?	1+, 2-, 2?
	Acoplamento Eferente	1+, 2-	?	?	1+, 2-, 2?
Coesão	Coesão das operações	6+, 3-	-	+	7+, 4-

Tabela 6.1: Propriedades e Métricas da Revisão Sistemática e Estudos de Caso

Conforme pode ser observado na Tabela 6.1, os estudos desenvolvidos nesse trabalho confimaram, junto com a maioria dos estudos da Revisão Sistemática, que POA é negativo para Tamanho do Vocabulário. Esse resultado mostra que POA aumenta o número de classes no código.

Outro ponto que pode ser observado na Tabela 6.1, é o fato de que uma métrica extraída dos estudos desse trabalho contraria a maioria dos estudos da Revisão Sistemática. Para a maioria dos estudos da Revisão Sistemática foi concluído que POA aumenta o número de linhas de código e para os dois estudos desse trabalho POA diminui LOC.

Os estudos empíricos realizados nesse trabalho contribuiram com a avaliação de grande quantidade de métricas em comparação com os estudos analisados na revisão sistemática. Além disso, diferentemente dos 13 artigos e 27 estudos empíricos, esse trabalho apresentou valores de atributos externos de qualidade como Eficiência, Manutenabilidade, Portabilidade, Confiabilidade e Usabilidade.

Outro diferencial desse trabalho em comparação com os estudos encontrados na Revisão Sistemática foi a implementação de um tipo de interesse transversal que não foi implementado como aspecto em nenhum dos 27 estudos: *logging*.

Este trabalho deu origem a dois artigos aceitos para publicação em *proceedings* de conferências internacionais em Engenharia de Software:

- "*A Systematic Review on Evaluation of Aspect Oriented Programming Using Software Metrics*" [França e Soares 2012], *International Conference on Enterprise Information Systems (ICEIS)*, Wroclaw - Polônia, Junho de 2012.
- "*An Empirical Evaluation of Refactoring Crosscutting Concerns into Aspects using Software Metrics*", *International Conference on Information Technology - New Generations (ITNG)*, Las Vegas - Estados Unidos, Abril de 2013.

Ambos artigos foram publicados nas *proceedings* de conferências internacionais qualificadas pela CAPES com conceito B1.

Ameaças à Validade

Os estudos experimentais, em geral, apresentam ameaças a validade [Yin 2003] [Runeson e Höst 2009]. As ameaças a validade desse trabalho são apresentadas a seguir.

Como a maioria dos estudos de caso, é difícil generalizar conclusões a partir de duas aplicações. Portanto, existem ameaças à validade externa. Os estudos de caso foram aplicados em softwares escolhidos respeitando critérios previamente estabelecidos. As duas aplicações utilizadas na refatoração com POA não foram escolhidos aleatoriamente, o que restringe a ameaça à validade externa. Além disso, as conclusões sobre qualidade de software com refatoração utilizando POA foram formuladas com base nos resultados obtidos nesse trabalho e nos estudos experimentais encontrados a partir da Revisão Sistemática.

A ameaça à validade interna do estudo consiste no fato de que foram utilizadas ferramentas que automaticamente extraem métricas do software. Tais métricas podem fornecer valores contabilizados erroneamente ou podem conter *bugs* que podem afetar a avaliação, de forma a fornecer conclusões erradas. Essa ameaça foi reduzida pela estratégia de usar duas ferramentas que produzem métricas em comum. Sendo assim, foi possível comparar se as métricas de duas ferramentas compartilham os mesmos resultados. Os resultados não precisam ser exatamente iguais, pois cada ferramenta possui uma forma padrão para apresentar seus resultados, mas os resultados precisam ser unânimes em favorecer um determinado paradigma.

Outra questão que pode ser considerada como ameaça é o fato de que a refatoração nos dois softwares foi desenvolvida por apenas uma pessoa. Uma pessoa diferente poderia escolher uma abordagem diferente para implementar interesses transversais como aspecto. A abordagem desse trabalho foi implementar como aspectos todos os interesses transversais identificados no código de dois softwares. Um interesse transversal adicional, o rastreamento, foi implementado para investigar o efeito de vários tipos de interesses transversais no código. Rastreamento foi o último interesse transversal implementado, portanto foi possível perceber a qualidade do código antes e depois de sua implementação. O efeito causado por este interesse transversal foi detalhadamente especificado durante o trabalho.

Os resultados das métricas foram avaliados com uma conclusão positiva, negativa ou inconclusiva para POA. Uma ameaça a validade dessas conclusões é que os resultados obtidos são avaliados independentemente do tamanho do estudo de caso.

Contribuições

As contribuições desse trabalho são:

- Revisão Sistemática que revelou pouca quantidade de estudos empíricos na avaliação de qualidade de software refatorado com POA.
- Dois estudos de casos abordando questões pouco exploradas na literatura:
  - Implementação de interesses transversais que não receberam atenção adequada na literatura;
  - Aplicação de grande quantidade de métricas;
  - Análises sobre o impacto de POA nos atributos de qualidade de software.
- Aplicação de ferramentas que realizam extração automática de métricas nos softwares;
- Avaliação dos benefícios de POA usando métricas.
- Publicação de dois artigos em conferências internacionais.

Trabalhos Futuros

Alguns trabalhos futuros podem ser realizados para melhorar a avaliação empírica de Programação Orientada a Aspectos. Como pôde ser observado, os poucos estudos empíricos ainda não foram suficientes para responder algumas questões. Alguns interesses transversais ainda não foram explorados de forma adequada, como *caching*, segurança, sincronização e controle de concorrência. Além disso, softwares de tamanho grande (maior que 30 KLOC) foram pouco considerados.

Como proposta de trabalhos futuros, pode-se listar:

- Produção de avaliação empírica de POA através da Refatoração de softwares de tamanhos maiores (acima de 30 mil linhas de código).
- Implementação de outros interesses transversais pouco considerados na literatura, como *caching*, segurança, sincronização e controle de concorrência.
- Avaliação do impacto da refatoração com POA nos atributos externos de qualidade, incluindo facilidade de compreensão, testabilidade, modularidade, complexidade, portabilidade, facilidade de uso, facilidade de reuso, eficiência e facilidade de aprendizado.
- Outra questão que deve ser abordada é a relação do código refatorado com sua arquitetura. A conformidade de arquitetura exige que o software implementado apresente os atributos de qualidade projetados na arquitetura. O problema é que em muitos casos a arquitetura descrita na documentação não é refletida no código fonte. Há divergências entre a Arquitetura Conceitual (existe na documentação do software, como-projetada) e a Arquitetura Concreta (derivada do código fonte, como-implementada) [Ducasse e Pollet 2009].

  A conformidade da arquitetura de softwares em POA pode ser verificada como proposta de pesquisa em trabalhos futuros.

Referências Bibliográficas

[AJD] AJDT. http://www.eclipse.org/ajdt/. Online; acessado em 21-Jan-2013.

[ATM] ATM. http://www.math-cs.gordon.edu/courses/cs211/ATMExample/. Online; acessado em 19-Set-2012.

[Ecl] Eclipse IDE (Integrated Development Environment). http://www.eclipse.org/. Online; acessado em 21-Dez-2012.

[ISO] ISO9126 : Information technology - Software Product Evaluation - Quality characteristics and guidelines for their use - 1991.

[JMo] JMoney. http://jmoney.sourceforge.net/. Online; acessado em 19-Set-2012.

[Plu] Plugin Metric for Eclipse. http://sourceforge.net/projects/metrics/. Online; acessado em 17-Set-2012.

[Qua] Qualitas Corpus. http://qualitascorpus.com/. Online; acessado em 19-Set-2012.

[Son] Sonar. http://www.sonarsource.org/. Online; acessado em 17-Set-2012.

[Win] WinMerge. http://winmerge.org/. Online; acessado em 07-Dez-2012.

[Int 2013] (2013). Definição de Interesses Transversais. http://msdn.microsoft.com/en-us/library/ee658105.aspx. Online; acessado em 03-Jan-2013.

[Ali et al. 2010] Ali, M. S., Ali Babar, M., Chen, L., e Stol, K.-J. (2010). A Systematic Review of Comparative Evidence of Aspect-Oriented Programming. *Information and Software Technology*, 52:871–887.

[Arnold et al. 2006] Arnold, K., Gosling, J., e Holmes, D. (2006). *The Java Programming Language*. Addison-Wesley, Boston, MA, USA, 4 edition.

[Bartsch e Harrison 2008] Bartsch, M. e Harrison, R. (2008). An Exploratory Study of the Effect of Aspect-Oriented Programming on Maintainability. *Software Quality Control*, 16:23–44.

[Booch et al. 2005] Booch, G., Rumbaugh, J., e Jacobson, I. (2005). *The Unified Modeling Language User Guide*. Addison-Wesley Professional, Boston, MA, USA, 2 edition.

[Bourque et al. 2002] Bourque, P., Dupuis, R., Abran, A., Moore, J. W., Tripp, L. L., e Wolff, S. (2002). Fundamental Principles of Software Engineering - A Journey. *Journal of Systems and Software*, 62(1):59–70.

[Cacho et al. 2008] Cacho, N., Filho, F. C., Garcia, A., e Figueiredo, E. (2008). EJFlow: Taming Exceptional Control Flows in Aspect-Oriented Programming. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development*, pp. 72–83.

[Cacho et al. 2006] Cacho, N., Sant'Anna, C., Figueiredo, E., Garcia, A., Batista, T., e Lucena, C. (2006). Composing Design Patterns: a Scalability Study of Aspect-Oriented Programming. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pp. 109–121.

[Ceccato e Tonella 2004] Ceccato, M. e Tonella, P. (2004). Measuring the Effects of Software Aspectization. In *First Workshop on Aspect Reverse Engineering*.

[Chidamber e Kemerer 1994] Chidamber, S. R. e Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

[Clayberg e Rubel 2008] Clayberg, E. e Rubel, D. (2008). *Eclipse Plug-ins*, volume 2 de *the Eclipse Series*. Addison-Wesley, 3 edition.

[Coelho et al. 2008] Coelho, R., Rashid, A., Garcia, A., Ferrari, F. C., Cacho, N., Kulesza, U., von Staa, A., e de Lucena, C. J. P. (2008). Assessing the Impact of Aspects on Exception Flows: An Exploratory Study. In *Proceedings of the European Conference on Object-Oriented Programming*, pp. 207–234.

[Colyer et al. 2005] Colyer, A., Clement, A., Harley, G., e Webster, M. (2005). *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Addison-Wesley, Upper Saddle River, NJ.

[Cui et al. 2009] Cui, Z., Wang, L., Li, X., e Xu, D. (2009). Modeling and Integrating Aspects with UML Activity Diagrams. In *Proceedings of the 2009 ACM Symposium On Applied Computing*, SAC '09, pp. 430–437.

[d'Amorim e Borba 2010] d'Amorim, F. e Borba, P. (2010). Modularity Analysis of Use Case Implementations. In *Brazilian Symposium on Software Components, Architectures and Reuse*, pp. 11–20.

[Deitel e Deitel 2007] Deitel, H. M. e Deitel, P. J. (2007). *Java How to Program*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 7th edition.

[Dijkstra 1997] Dijkstra, E. W. (1997). *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

[Ducasse e Pollet 2009] Ducasse, S. e Pollet, D. (2009). Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591.

[Figueiredo et al. 2008] Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., e Dantas, F. (2008). Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proceedings of the 30th International Conference on Software Engineering*, pp. 261–270.

[Filman et al. 2005] Filman, R. E., Elrad, T., Clarke, S., e Akşit, M. (2005). *Aspect-Oriented Software Development*. Addison-Wesley, Boston.

[Fowler 1999] Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

[França e Soares 2012] França, J. M. S. e Soares, M. S. (2012). A Systematic Review on Evaluation of Aspect Oriented Programming Using Software Metrics. In *Proceedings of the 14th International Conference on Enterprise Information Systems (ICEIS)*, pp. 77–83.

[Greenwood et al. 2007] Greenwood, P., Bartolomei, T. T., Figueiredo, E., Dósea, M., Garcia, A. F., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U., e Rashid, A. (2007). On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *Proceedings of the European Conference on Object-Oriented Programming*, pp. 176–200.

[Hanenberg et al. 2009] Hanenberg, S., Kleinschmager, S., e Josupeit-Walter, M. (2009). Does Aspect-Oriented Programming Increase the Development Speed for Crosscutting Code? An Empirical Study. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pp. 156–167. IEEE Computer Society.

[Henderson-Sellers 1996] Henderson-Sellers, B. (1996). *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Hitz e Montazeri 1995] Hitz, M. e Montazeri, B. (1995). Measuring Coupling and Cohesion in Object-Oriented Systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, pp. 412–421.

[Hoffman e Eugster 2008] Hoffman, K. e Eugster, P. (2008). Towards Reusable Components with Aspects: An Empirical Study on Modularity and Obliviousness. In *Proceedinds of the 30th International Conference on Software Engineering*, pp. 91–100.

[Hovsepyan et al. 2010] Hovsepyan, A., Scandariato, R., Van Baelen, S., Berbers, Y., e Joosen, W. (2010). From Aspect-Oriented Models to Aspect-Oriented Code?: The Maintenance Perspective. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pp. 85–96.

[Javed et al. 2007] Javed, A. Z., Strooper, P. A., e Watson, G. N. (2007). Automated Generation of Test Cases Using Model-Driven Architecture. In *Proceedings of the Second International Workshop on Automation of Software Test*, AST '07, pp. 3–9.

[Keith e Schincariol 2009] Keith, M. e Schincariol, M. (2009). *Pro JPA 2: Mastering the Java Persistence API*. Apress, Berkely, CA, USA.

[Kiczales et al. 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., e Irwin, J. (1997). Aspect-Oriented Programming. volume 1241, chapter 10, pp. 220–242.

[Kitchenham 2004] Kitchenham, B. (2004). Procedures for Performing Systematic Reviews. Technical report, Department of Computer Science, Keele University, UK.

[Koscianski e Soares 2007] Koscianski, A. e Soares, M. S. (2007). *Qualidade de Software*. Novatec Editora, São Paulo, SP, Brasil, 2 edition.

[Kouskouras et al. 2008] Kouskouras, K. G., Chatzigeorgiou, A., e Stephanides, G. (2008). Facilitating Software Extension with Design Patterns and Aspect-Oriented Programming. *Journal of Systems and Software*, 81:1725–1737.

[Laddad 2003] Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., USA, first edition.

[Laddad 2009] Laddad, R. (2009). *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications Co., Greenwich, CT, USA, second edition.

[Madeyski e Szala 2007] Madeyski, L. e Szala, L. (2007). Impact of Aspect-Oriented Programming on Software Development Efficiency and Design Quality: an Empirical Study. *IET Software*, 1(5):180–187.

[Malta e Valente 2009] Malta, M. N. e Valente, M. T. O. (2009). Object-Oriented Transformations for Extracting Aspects. *Information and Software Technology*, 51(1):138–149.

[Martin 1994] Martin, R. (1994). OO Design Quality Metrics - An Analysis of Dependencies. In *Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*.

[McCabe 1976] McCabe, T. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, pp. 308–320.

[Mortensen et al. 2012] Mortensen, M., Ghosh, S., e Bieman, J. M. (2012). Aspect-Oriented Refactoring of Legacy Applications: An Evaluation. *IEEE Transactions on Software Engineering*, 38:118–140.

[Parnas 1972] Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM,*, 15(12):1053–1058.

[Pouria Shaker 2005] Pouria Shaker, D. K. P. (2005). An Introduction to Aspect-Oriented Software Development .

[Pressman 2011] Pressman, R. S. (2011). *Engenharia de Software - Uma abordagem Profissional*. MCGRAW-Hill, Porto Alegre, RS, Brasil, 7 edition.

[Przybylek 2010] Przybylek, A. (2010). What is Wrong with AOP? In *International Joint conference on Software Technologies (ICSOFT(2))*, pp. 125–130.

[Przybylek 2011] Przybylek, A. (2011). Impact of Aspect-Oriented Programming on Software Modularity. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, The European Conference on Software Maintenance and Reengineering (CSMR '11), pp. 369–372.

[Ramirez et al. 2011] Ramirez, A. J., Jensen, A. C., e Cheng, B. H. (2011). An Aspect-Oriented Approach for Implementing Evolutionary Computation Applications. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*, pp. 153–164.

[Richards 2006] Richards, M. (2006). *Java Transaction Design Strategies*. InfoQ Enterprise Software Development Series. Lulu.com.

[Runeson e Höst 2009] Runeson, P. e Höst, M. (2009). Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering.*, 14(2):131–164.

[Sant'anna et al. 2003] Sant'anna, C., Garcia, A., Chavez, C., Lucena, C., e v. von Staa, A. (2003). On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In *Proceedings XVII Brazilian Symposium on Software Engineering*, pp. 19–34.

[Shaw 2002] Shaw, M. (2002). What Makes Good Research in Software Engineering? *International Journal on Software Tools for Technology Transfer - STTT*, 4(1):1–7.

[Sommerville 2010] Sommerville, I. (2010). *Software Engineering*. Addison Wesley, Essex,UK, 9 edition.

[Tizzei et al. 2011] Tizzei, L. P., Dias, M. O., Rubira, C. M. F., Garcia, A., e Lee, J. (2011). Components Meet Aspects: Assessing Design Stability of a Software Product Line. *Information & Software Technology*, 53(2):121–136.

[Wirth 1971] Wirth, N. (1971). Program Development by Stepwise Refinement. *Communications of the ACM*, 14:221–227.

[Wong et al. 2011] Wong, W. E., Tse, T. H., Glass, R. L., Basili, V. R., e Chen, T. Y. (2011). An Assessment of Systems and Software Engineering Scholars and Institutions (2003-2007 and 2004-2008). *Journal of Systems and Software*, 84(1):162–168.

[Yin 2003] Yin, R. K. (2003). *Case Study Research. Design and Methods*, volume 5 de *Applied Social Research Method Series*. Sage Publications, California, USA, third edition.