UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE SISTEMAS DE INFORMAÇÃO – BACHARELADO

ANÁLISE COMPARATIVA ENTRE PROGRAMAÇÃO ORIENTADA A OBJETOS E ORIENTADA A ASPECTOS

KELLI APARECIDA BEZ BATTI DA SILVA

BLUMENAU
2005

2005/2-13

KELLI APARECIDA BEZ BATTI DA SILVA

ANÁLISE COMPARATIVA ENTRE PROGRAMAÇÃO ORIENTADA A OBJETOS E ORIENTADA A ASPECTOS

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Sistemas de Informação — Bacharelado.

Prof. Mauro Marcelo Mattos , Dr. - Orientador

BLUMENAU
2005

2005/2-13

ANÁLISE COMPARATIVA ENTRE PROGRAMAÇÃO ORIENTADA A OBJETOS E ORIENTADA A ASPECTOS

Por

KELLI APARECIDA BEZ BATTI DA SILVA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

	_____
Presidente:	Prof. Mauro Marcelo Mattos , Dr. – Orientador, FURB
	_____
Membro:	Prof. Everaldo Artur Grahl, Mestre – FURB
	_____
Membro:	Prof. Alexander Roberto Valdameri, Mestre – FURB

Blumenau, 08 de dezembro de 2005

Dedico este trabalho aos meus pais, aos meus amigos e ao meu orientador.

AGRADECIMENTOS

Agradeço a Deus, pela oportunidade de concluir mais uma etapa em minha vida.

Aos meus pais e irmão, por tudo o que fizeram e ainda fazem por mim.

Aos meus amigos, por compreenderem a minha ausência durante o desenvolvimento deste trabalho e por todo o apoio, atenção e confiança que me foram concedidos vezes a fio.

À professora Fabiane Barreto Vavassori Benitti, pelos questionamentos que colaboraram para o progresso do trabalho.

Em especial a Gilmar José Purin, por me apresentar a orientação a aspectos.

Ao meu orientador, Mauro Marcelo Mattos, pelo entusiasmo no desenvolvimento deste trabalho desde as nossas conversas iniciais, pela dedicação e amizade que resultaram no sucesso do trabalho.

O que sabemos é uma gota, o que ignoramos é um oceano.

Isaac Newton

RESUMO

Este trabalho apresenta uma análise comparativa entre Programação Orientada a Objetos (*Object Oriented Programming*) e Programação Orientada a Aspectos (*Aspect Oriented Programming*) no desenvolvimento de requisitos ortogonais de software, aplicando ambos em um estudo de caso. Para a realização da análise, as duas implementações – POO e POA – foram avaliadas através dos critérios de performance, volume de código, tempo de desenvolvimento e funcionalidade da versão final. O comparativo demonstrou a eficiência do uso da programação orientada a aspectos na implementação de requisitos ortogonais.

Palavras-chave: Requisitos ortogonais. Programação orientada a objetos. Programação orientada a aspectos.

ABSTRACT

This work presents a comparative study between Object Oriented Programming (OOP) and Aspect Oriented Programming (AOP) in the development of software crosscutting concerns, applying both in a case study. To accomplish the study, both implementations – OOP and AOP – were compared through issues such as performance, lines of code, development time and functionality of the final version. The comparative study shows the efficiency of aspect oriented programming in the development of crosscutting concerns.

Key-words: Crosscutting concerns. Object oriented programming. Aspect oriented programming.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

AOP – *Aspect Oriented Programming*

B2B - *Business to Business*

GRASP – *General Responsibility Assignment Software Patterns*

GoF – *Gang of Four*

JIT – *Just In Time*

JVM – *Java Virtual Machine*

MLOC – *Method Lines of Code*

MVC – *Model-View-Controller*

OAB – Ordem dos Advogados do Brasil

OOP – *Object Oriented Programming*

POA – Programação Orientada a Aspectos

POO – Programação Orientada a Objetos

UML – *Unified Modeling Language*

XML - *Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Em sistemas computacionais complexos, é comum a existência de responsabilidades de interesse comum a vários módulos como, por exemplo, requisitos não funcionais de gerenciamento de dados, *logging* de transações, controle de concorrência, tratamento de exceções, autenticação, autorização, entre outros. Denominados interesses ortogonais, muitas vezes são difíceis de isolar porque demandam alterações em vários pontos do sistema, ocasionando entrelaçamento e espalhamento de código.

Segundo Chung e Mylopoulos (1999 apud SANTOS et al, 2000, p. 210), os requisitos não funcionais, por sua natureza, são difíceis de serem tratados durante o projeto e a implementação e também são difíceis de validar, freqüentemente são descritos de forma breve e ambígua, interagem com outros requisitos não funcionais e possuem um impacto global no sistema.

Impactos de natureza ortogonal são de ampla influência no sistema, e implementados de forma que cada parte afetada do sistema contenha código de lógica com várias áreas de interesse, resultam na perda de coesão e aumento do acoplamento dos módulos afetados. (LARMAN, 2004, p.490).

> Quando a OOP foi difundida pelo mercado desenvolvedor de software, houve um efeito dramático na maneira com que os sistemas eram desenvolvidos. Desenvolvedores podiam visualizar sistemas como grupos de entidades e as interações entre essas entidades, o que permitia o desenvolvimento de sistemas maiores e mais complexos em menos tempo. O único problema com a OOP é ser essencialmente estática, e mudanças nos requisitos podem provocar profundos impactos de desenvolvimento. (O'REGAN, 2004, p. 1, tradução nossa)

A Programação Orientada a Objetos (POO) é a metodologia mais comum empregada para o desenvolvimento de interesses do domínio do sistema, porém limitada para muitos interesses ortogonais, especialmente em sistemas complexos. (LADDAD,2003, p. 4, tradução nossa).

A Programação Orientada a Aspectos (POA) é uma nova metodologia que provê a

separação dos interesses ortogonais, introduzindo uma nova unidade de modularização – o Aspecto (LADDAD, 2003, p. 4, tradução nossa).

Tendo por objetivo validar esta metodologia, este trabalho apresenta um estudo comparativo do emprego de programação orientada a objetos e programação orientada a aspectos. Para caracterizar um objeto de análise, foi desenvolvida uma aplicação orientada a objetos para o gerenciamento básico de processos jurídicos. A seguir, esta aplicação foi estendida para contemplar a inclusão de um requisito não funcional de interesse ortogonal, ainda utilizando somente programação orientada a objetos. Posteriormente, este mesmo requisito não funcional foi agregado à aplicação usando-se programação orientada a aspectos.

Durante a análise comparativa, as duas implementações – POO e POA – foram avaliadas através dos critérios performance, volume de código, tempo de desenvolvimento e funcionalidade da versão final.

1.1 OBJETIVOS DO TRABALHO

O objetivo principal deste trabalho é realizar uma análise comparativa entre as abordagens de desenvolvimento de software orientado a objetos e orientado a aspectos na implementação de responsabilidades que são de interesse comum a vários módulos, os requisitos ortogonais (*crosscuting concerns*).

Os objetivos específicos do trabalho são:

a) especificar uma aplicação em duas versões de modo que a segunda versão seja considerada uma “manutenção” da primeira;
b) implementar a primeira e a segunda versão utilizando POO;
c) reimplementar a segunda versão utilizando POA.
d) identificar os requisitos a serem avaliados.

e) realizar a análise comparativa.

1.2 RELEVÂNCIA DO TRABALHO

A orientação a aspectos está gradativamente sendo popularizada, de forma muito positiva, quanto as suas contribuições na implementação de requisitos ortogonais de software.

A relevância principal deste trabalho é a adoção desta metodologia, no sentido de apresentá-la como alternativa para resolver necessidades comuns do desenvolvimento de sistemas, que muitas vezes se tornam atividades críticas.

Para muitos pesquisadores e profissionais (LADDAD, GRADECKI, LESIECK, etc..) , é cada vez mais notável a vantagem desta metodologia para superar deficiências da orientação a objetos, o que comprova, no mínimo, a importância da programação orientada a aspectos.

1.3 ESTRUTURA DO TRABALHO

Este trabalho está dividido em dez capítulos.

No primeiro capítulo, encontra-se uma introdução e os objetivos a serem alcançados com o desenvolvimento deste trabalho.

No segundo capítulo é apresentada a fundamentação teórica dos recursos utilizados para a realização deste trabalho.

O terceiro capítulo contextualiza o estudo de caso que possibilitou a análise comparativa entre as abordagens de desenvolvimento aplicadas.

No quarto capítulo são apresentados a especificação e o desenvolvimento do cenário inicial do estudo de caso

O quinto capítulo descreve um novo cenário para o estudo de caso, em virtude da

adição de novos requisitos não funcionais que não haviam sido previstos na primeira versão do sistema.

O sexto capítulo apresenta a especificação e a implementação dos novos requisitos não funcionais de interesse ortogonal que devem ser contemplados pela aplicação original, utilizando somente programação orientada a objetos.

Assim como o sexto capítulo, o sétimo capítulo apresenta a especificação e a implementação destes mesmos requisitos, contudo, abordando uma alternativa de desenvolvimento orientado a aspectos.

No oitavo capítulo, é descrito o mecanismo de tratamento de exceções, adicionalmente desenvolvido na terceira versão da aplicação, utilizando recursos da programação orientada a aspectos.

A análise comparativa entre POO e POA é apresentada no capítulo nove e tem por objetivo avaliar a implementação dos requisitos não funcionais de interesse ortogonal que foram agregados à aplicação original. Neste capítulo, são abordados os critérios performance, volume de código, tempo de desenvolvimento e funcionalidade da versão final.

Por fim, o capítulo dez trata as considerações finais e sugestões de extensões deste trabalho para projetos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

"Programar é divertido, mas desenvolver software de qualidade é difícil. Entre ótimas idéias, requisitos ou visão e um produto de software que funcione, existe muito mais do que simplesmente programar." (LARMAN, 2004, p.vii).

Neste capítulo são apresentados os principais conceitos relacionados ao contexto do projeto.

a) orientação a objetos;

b) padrões de projeto;

c) requisitos ortogonais de software;

d) orientação a aspectos;

e) AspectJ;

f) trabalhos correlatos.

2.1 ORIENTAÇÃO A OBJETOS

O famoso ditado popular "comprar um martelo não transforma você em um arquiteto" é bastante aplicado à essência da orientação a objetos. Desenvolver sistemas orientados a objetos não significa adotar somente uma linguagem de programação, é uma concepção diferente sobre sistemas computacionais.

> OO é um termo geral que inclui qualquer estilo de desenvolvimento que seja baseado no conceito de objeto – uma entidade que exibe características e comportamentos. Você pode aplicar uma estratégia orientada a objetos na programação, na análise e no projeto. (SINTES, 2002, p. 4).

Segundo Sintes (2002), "Um objeto é uma construção de software que encapsula estado e comportamento. Os objetos permitem que você modele seu software em termos reais e abstrações."

Segundo Leite e Rahal Junior (2002), “se considerarmos a Orientação ao Objeto como um novo paradigma de desenho de software, devemos considerar, também, uma nova maneira de pensar, porque apesar de a escrita do código continuar procedural, alguns conceitos mudam radicalmente: a estruturação e o modelo computacional”.

No contexto do projeto, a orientação a objetos foi utilizada no desenvolvimento da primeira e segunda versão da aplicação alvo da avaliação.

2.2 PADRÕES DE PROJETOS

Segundo Larman (2004), a noção formal de padrões originou-se dos padrões arquiteturais de Cristhoper Alexander. Os padrões para software tiveram início na década de 1980, com Kent Beck, que tomou conhecimento do trabalho de Alexander sobre padrões em arquitetura, e foram desenvolvidos por Beck e Ward Cunningham

> Na tecnologia de objetos, um **padrão** é uma descrição nomeada de um problema e uma solução que pode ser aplicada em novos contextos; em termos ideais, ele fornece orientação sobre sua aplicação em circunstâncias variáveis e leva em conta as forças e as soluções de compromisso. Muitos padrões fornecem orientação sobre a atribuição de responsabilidades a objetos, dada uma categoria específica de problemas. (LARMAN, 2004, p. 230, grifo do autor).

Deitel (2003) afirma que a utilização de padrões de projeto beneficia os desenvolvedores de sistemas a construir *software* confiável com arquiteturas comprovadas e experiência acumulada das empresas, promove a reutilização de projeto em sistemas futuros, ajuda a identificar armadilhas comuns no desenvolvimento de sistemas, ajuda a projetar sistemas de forma independente da linguagem na qual serão implementados, estabelece um vocabulário comum de projeto entre desenvolvedores e possivelmente encurta a fase de projeto em um processo de desenvolvimento de software.

2.2.1 Padrões de projeto GRASP

Um conjunto de padrões de projeto bastante difundido e aplicado é o *General Responsability Assignment Software Patterns* (GRASP). Segundo Larman (2004), "Os padrões GRASP ajudam a compreender o projeto essencial de objetos e aplicam o raciocínio de projeto de um modo metódico, racional e explicável. Essa abordagem à compreensão e ao uso dos princípios de projeto está baseada em *padrões de atribuição de responsabilidades*."

Os principais padrões do GRASP são:

a) especialista na informação;
b) criador;
c) coesão alta;
d) acoplamento fraco;
e) controlador.

Neste trabalho foram utilizados os padrões de projeto especialista na informação, coesão alta, acoplamento fraco e controlador. As seções seguintes descrevem as soluções propostas por estes quatro padrões segundo Larman (2004).

2.2.1.1 Padrão de projeto especialista na informação

Atribuir uma responsabilidade ao especialista na informação: a classe que tem a informação necessária para satisfazer a responsabilidade.

2.2.1.2 Padrão de projeto coesão alta

Atribuir uma responsabilidade de forma que a coesão permaneça alta. A coesão mede

o quanto as responsabilidades de um elemento são fortemente relacionadas.

2.2.1.3 Padrão de projeto acoplamento fraco

Atribuir uma responsabilidade de maneira que o acoplamento permaneça fraco. O acoplamento mede o quanto um elemento está conectado a, tem conhecimento de ou depende de muitos outros elementos.

2.2.1.4 Padrão de projeto controlador

Atribuir a responsabilidade de receber ou tratar uma mensagem de um evento do sistema a uma classe que represente uma das seguintes escolhas: represente todo o sistema, subsistemas, ou represente um cenário de um caso de uso dentro do qual ocorra um evento do sistema.

2.2.2 Padrões de projeto GoF

Outro importante conjunto de padrões de projeto é o GoF (*Gang of Four*). Os 23 padrões deste conjunto são classificados em criacionais, estruturais e comportamentais.

Nas sub-seções seguintes estão descritos os três padrões utilizados neste trabalho.

2.2.2.1 Padrão de projeto MVC (Model, View, Controller)

Segundo Gamma (1994), o padrão de projeto MVC tem por objetivo dividir os elementos de um sistema em três tipos de objetos: Modelo (*Model*), Controlador (*Controller*)

e Visão (*View*).

Os objetos de Modelo consistem no domínio do sistema, a Visão na interface apresentada ao usuário e o Controlador na maneira com que a interface reage aos eventos do usuário. Esta abordagem, segundo Gamma (1994), promove uma maior flexibilidade e reuso dos objetos.

2.2.2.2 Padrão de projeto Singleton

O padrão de projeto s*ingleton,* conforme define Gamma (1994), tem por objetivo garantir que uma classe possua somente uma instância e forneça um ponto global de acesso a esta instância.

2.2.2.3 Padrão de projeto Facade

O padrão *facade*, de acordo com Larman (2004), define um único ponto de contato para um subsistema – um objeto fachada que empacote o subsistema. Este objeto fachada apresenta uma interface unificada e é responsável por colaborar com os componentes do subsistema.

2.3 REQUISITOS ORTOGONAIS DE SOFTWARE

Sistemas de software são compostos por áreas de interesses ou responsabilidades distintas como, por exemplo, responsabilidades funcionais (lógica de negócio) e não-funcionais (performance, persistência de dados, *logging*, autenticação de usuários, segurança, verificação de erros, etc.). (DEXTRA, 2005).

"A atribuição competente de responsabilidades é extremamente importante no projeto orientado a objetos". (LARMAN, 2004, p. 231). Projetar um sistema através da separação de responsabilidades é fundamental para um projeto orientado a objetos, onde uma classe – especificação de objeto - é uma dimensão para a decomposição de responsabilidades. Contudo, algumas responsabilidades não são tão facilmente decompostas em apenas uma dimensão.

A programação orientada a objetos permite a realização de responsabilidades específicas de forma modularizada, através do encapsulamento de dados (estado) e operações (comportamento) em uma unidade denominada Objeto. Porém, quando uma responsabilidade não é específica, surge então a dificuldade em desenvolvê-la individualmente, em isolar esta responsabilidade em um objeto, ou até mesmo em um conjunto de objetos. Estas responsabilidades não individuais que afetam outras responsabilidades são denominadas requisitos ortogonais de software.

Estes requisitos representam situações em que uma responsabilidade do sistema é encontrada em vários objetos, porém esta responsabilidade não está diretamente relacionada aos objetivos definidos para estes objetos.

Para explanar mais claramente o que são os requisitos ortogonais, a seguir é apresentada uma análise de dois requisitos de software utilizados no objeto alvo da análise:

a) o sistema deve permitir o cadastro de pessoas.
b) o sistema deve prover o mecanismo de controle de acesso às funcionalidades do sistema. Este mecanismo deverá ser implementado nas operações de sistema e nas telas do sistema.

No requisito "a", o objetivo é permitir o registro de pessoas no sistema. Portanto, em um sistema orientado a objetos, uma abordagem seria a criação de uma classe chamada Pessoa com a responsabilidade de manter o estado e o comportamento de uma pessoa do

mundo real, encapsulando o código necessário para o cadastro de uma pessoa.

No requisito de software “b”, o objetivo é garantir que apenas usuários com os devidos privilégios possam executar as funcionalidades do sistema. Desta forma, o cadastro de pessoa, bem como todos os outros cadastros e funcionalidades do sistema, deverão garantir que o requisito de autorização seja satisfeito, ou seja, que executarão apenas se o usuário estiver autorizado para tal operação. Neste caso, haverá código espalhado pelas funcionalidades do sistema para satisfazer o objetivo de autorização, descaracterizando o encapsulamento e a responsabilidades atribuídas aos objetos.

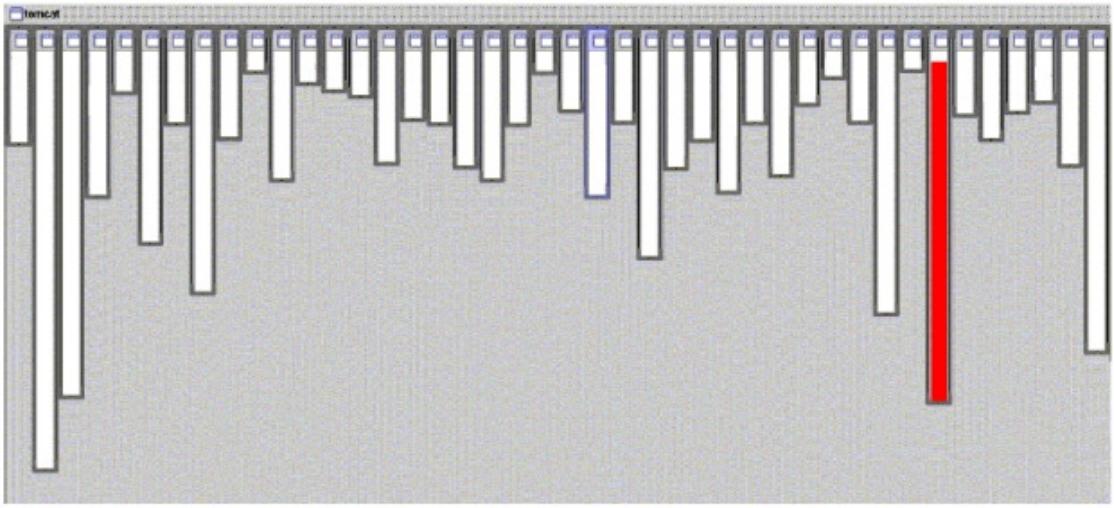
A classe Pessoa, por exemplo, que originalmente foi desenvolvida para manter o estado e o comportamento e uma pessoa do mundo real, receberia também a responsabilidade de verificar se o usuário autenticado está autorizado a registrar uma pessoa no sistema. Este cenário afeta a coesão da classe Pessoa. Segundo o padrão de projeto **coesão alta**, é importante evitar que classes se tornem complexas demais por assumirem demasiadas responsabilidades.

Os problemas aumentariam significativamente se o cenário descrito fosse um pouco maior, com mais alguns requisitos ortogonais de software:

a) o sistema deve manter um histórico de execução dos cadastros em uma tabela de banco de dados, armazenando o usuário que executou o cadastro, o horário e a data de execução para fins de histórico.
b) o sistema deve registrar em um arquivo de *log* todos os erros que ocorrerem durante a execução de todos os processos do sistema, para fins de identificação de problemas e estimativas quantitativas e qualitativas.
c) o sistema deve monitorar o tempo de execução de todos os processos do sistema, para fins de avaliação de performance.

Para caracterizar o problema em um contexto mais complexo, Clement e Kersten

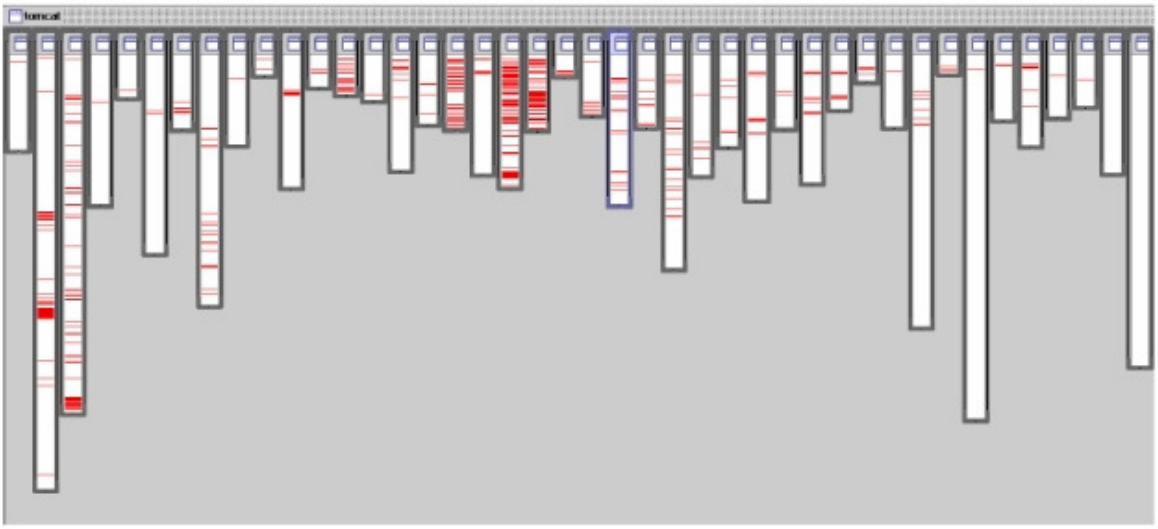
(2005) apresentam na Figura 1 a implementação do processo de validação (*parsing*) de XML do Apache Tomcat. A coluna em cor cinza demonstra as linhas de código relevantes deste procedimento, modularizado em uma apenas uma região.



Fonte: Clement, Kersten (2005, p. 3)

Figura 1 – Implementação do processo de validação (*parsing*) de XML no Apache Tomcat.

A Figura 2 apresenta a implementação do processo de *loggin* do Apache Tomcat. As marcações acinzentadas nas colunas demonstram as linhas de código relevantes deste procedimento, espalhado por várias regiões.



Fonte: Clement, Kersten (2005, p. 5)

Figura 2 – Implementação do processo de *loggin* do Apache Tomcat.

2.4 ORIENTAÇÃO A ASPECTOS

A orientação a aspectos demanda um pensamento diferenciado sobre projeto e implementação de sistemas. Segundo Laddad (LADDAD, 2003, p.30, tradução nossa), um principio fundamental da engenharia de software é que a separação de interesses resulta em um sistema melhor compreendido e mais facilmente mantido. Enquanto interesses ortogonais, que afetam várias partes de um sistema, são normalmente independentes um dos outros, a utilização de alternativas da POO leva a uma implementação que não preserva essa independência.

O objetivo da orientação a aspectos não é substituir a orientação a objetos, mas suportar a separação de interesses em um sistema. (GRADECKI; LESIECKI, 2003, p. 10, tradução nossa).

A orientação a aspectos oferece uma metodologia de implementação de interesses ortogonais de forma genérica e metódica, modularizando-os através do encapsulamento dentro de uma nova unidade denominada Aspecto. Esta metodologia complementa a orientação a objetos introduzindo uma nova dimensão para a decomposição das responsabilidades.

> Apresentamos uma análise de por que algumas decisões de projeto sempre foram muito difíceis de serem capturadas no código. Chamamos de aspectos as questões tratadas por essas decisões e mostramos que a razão pela qual sempre foram difíceis de serem capturadas é que elas atravessam (*crosscut*) a funcionalidade básica do sistema. Apresentamos a base para uma nova técnica de programação, chamada de programação orientada a aspectos (POA), que possibilita expressar claramente programas que envolvem esses aspectos, incluindo o isolamento, a composição e a reutilização apropriados do código do aspecto. (KICZALES, 1997 apud CHAVEZ, 2004, p. 50).

2.4.1 Elementos básicos da orientação a aspectos

A orientação a aspectos possui alguns elementos básicos descritos a seguir. Esta seção

baseia-se na explanação destes conceitos por Chavez (2004) e Tirelo (2004).

- componente: unidade funcional expressa em uma linguagem de componentes; Propriedades de um sistema, no qual a implementação pode ser encapsulada de forma limpa em um procedimento generalizado.
- aspecto: unidade não funcional expressa em uma linguagem de aspectos; propriedades de um sistema, no qual a implementação não pode ser encapsulada em um procedimento generalizado.
- *crosscuting*: entrelaçamento entre os interesses de domínio e os interesses ortogonais.
- pontos de junção (*join points*): pontos definidos na execução de um programa.
- conjuntos de junção (*point cuts*): elementos da semântica da linguagem de componentes com os quais os programas de aspectos coordenam; elementos compostos por pontos de junção que tem por objetivo reunir informações a respeito do contexto dos pontos selecionados.
- regras de junção: código relativo aos requisitos ortogonais que deve ser executado nos pontos de junção.
- processo de combinação (*weaving*): composição entre os componentes e os aspectos.
- combinador de aspectos (*aspect weaver*): processador de linguagem especial que oferece suporte à composição entre os componentes e os aspectos.

2.4.2 Metodologia da orientação a aspectos

Segundo Laddad (2003), a metodologia da orientação a aspectos define três etapas:

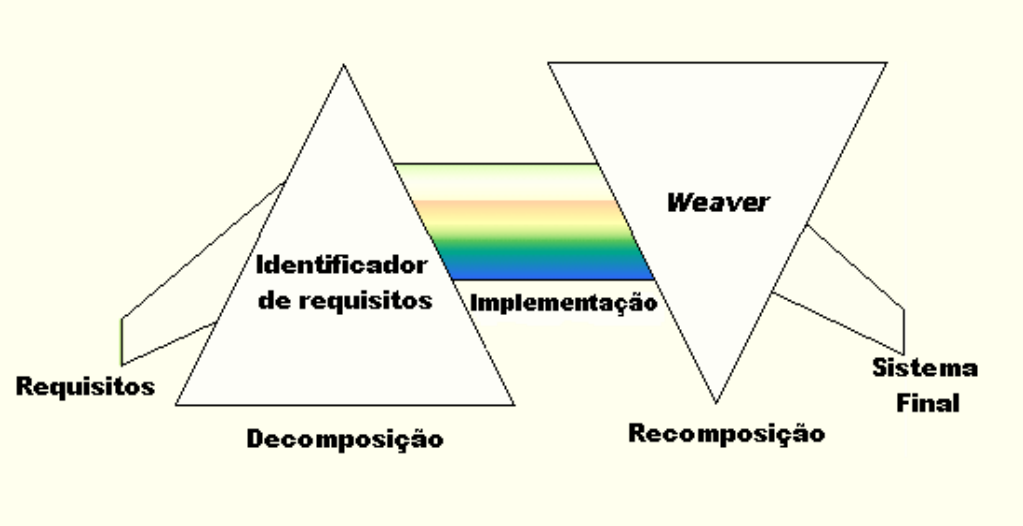
a) decomposição: os requisitos do sistema são identificados e classificados como

interesses de domínio ou interesses ortogonais;

b) implementação: os interesses são implementados separadamente. Os interesses de domínio são implementados em classes e os interesses ortogonais em aspectos;

c) recomposição: o sistema é recomposto com base nas regras de recomposição que foram definidas nos aspectos. Esta fase é denominada "*weaving*", processo no qual as classes e os aspectos são entrelaçados.

A Figura 3 ilustra o processo de decomposição, implementação e recomposição dos requisitos.



Fonte: Adaptado de Laddad (2003, p. 22)

Figura 3 – Metodologia da Orientação a Aspectos

Segundo Larman (2004), as tecnologias orientadas a aspectos apóiam a pós-compilação, entrelaçando interesses ortogonais no código de maneira transparente para o desenvolvedor. Essas estratégias mantêm a ilusão de separação durante o trabalho de desenvolvimento e recompõem o interesse antes da execução.

Alguns exemplos de aspectos, que mantêm interesses ortogonais, são gerenciamento de dados, *logging* de transações, controle de concorrência, tratamento de exceções, autenticação, autorização, entre outros.

2.4.3 Benefícios da orientação a aspectos

Segundo Laddad (2003), a orientação a aspectos demanda tempo, paciência e prática. Porém, seus benefícios são mais relevantes do que seus custos. Os benefícios propostos por Laddad com a adoção desta metodologia são:

a) responsabilidades bem definidas através de módulos individuais: permite a um módulo ter somente a responsabilidade relacionada ao seu objetivo principal; um módulo não mais está sujeito a interesses ortogonais (*crosscuting concerns*).

b) maior modularização: fornece um mecanismo para endereçar cada requisito individualmente com o menor acoplamento possível.

c) facilita a evolução dos sistemas: no surgimento de novos requisitos ortogonais, será suficiente a inclusão de novos aspectos, sem alterar o domínio do sistema. No surgimento de novas funcionalidades de domínio, os aspectos existentes, se desenvolvidos de forma adequada, serão entrelaçados com estas novas funcionalidades sem qualquer alteração, fazendo com que a evolução dos sistemas que utilizam a metodologia sejam coerentes.

d) possibilita a postergação de decisões arquiteturais: está relacionado ao dilema dos arquitetos de software frente à diversidade de preocupações existentes em sistemas computacionais. A metodologia permite aos arquitetos postergar decisões de arquitetura do software porque é possível desenvolvê-las como aspectos separados do domínio do sistema, sem causar alterações críticas. Neste caso, os arquitetos podem focar o que for necessário no momento correto.

e) aumenta a produtividade no desenvolvimento: a existência de responsabilidades bem definidas através de módulos individuais torna o trabalho dos desenvolvedores mais produtivo. Reutilização de código reduz o tempo de

desenvolvimento. A facilidade da evolução dos sistemas diminui a responsabilidade de novos requisitos. Todos estes benefícios induzem ao desenvolvimento e teste de forma mais produtiva.

f) diminui custos de implementação: ao evitar o custo de modificações em diversos módulos na implementação de requisitos ortogonais.

2.5 ASPECTJ

"AspectJ é uma simples e prática extensão à linguagem Java, orientada a aspectos." (KICZALES, 2001, p. 1, tradução nossa). Esta linguagem consiste em duas partes: especificação da linguagem e implementação da linguagem. A especificação da linguagem define a linguagem para escrita do código fonte dos aspectos, utilizando a linguagem Java e as extensões do AspectJ para a implementação dos pontos de combinação dos requisitos ortogonais. A implementação da linguagem fornece as ferramentas necessárias à compilação, *debugging*, e a integração com os ambientes de desenvolvimento.

2.5.1 Especificação da linguagem.

Esta seção é baseada em (LADDAD, 2003) e condensa os elementos básicos da linguagem de forma a facilitar a compreensão dos principais conceitos utilizados no trabalho.

2.5.1.1 Aspect

Em AspectJ, a principal unidade é chamada de *aspect*. Um *aspect* define uma responsabilidade específica que pode afetar várias partes de um sistema. Assim como uma

classe Java, um *aspect* pode definir atributos, métodos e uma hierarquia de *aspects*.

Os *aspects* implementam as regras de combinação (*weaving rules*) do compilador, que são denominadas *crosscutings* e classificadas em *static crosscuting* e *dynamic crosscuting*.

As regras classificadas como *"static crosscuting"* alteram a estrutura estática de classes, adicionando membros (atributos, métodos e construtores), alterando a hierarquia das classes e convertendo uma exceção checada para uma não checada.

Os *dynamic crosscuting* alteram ou adicionam comportamentos à execução de programas, afetando sua estrutura dinâmica através da intercepção de pontos no fluxo de execução de um programa.

Um *aspect* é composto por *join points*, *point cuts*, *advices*, *introductions* e adicionalmente por elementos de uma típica classe Java. A Figura 4 apresenta o código fonte de um *aspect*, delimitando as suas principais áreas.

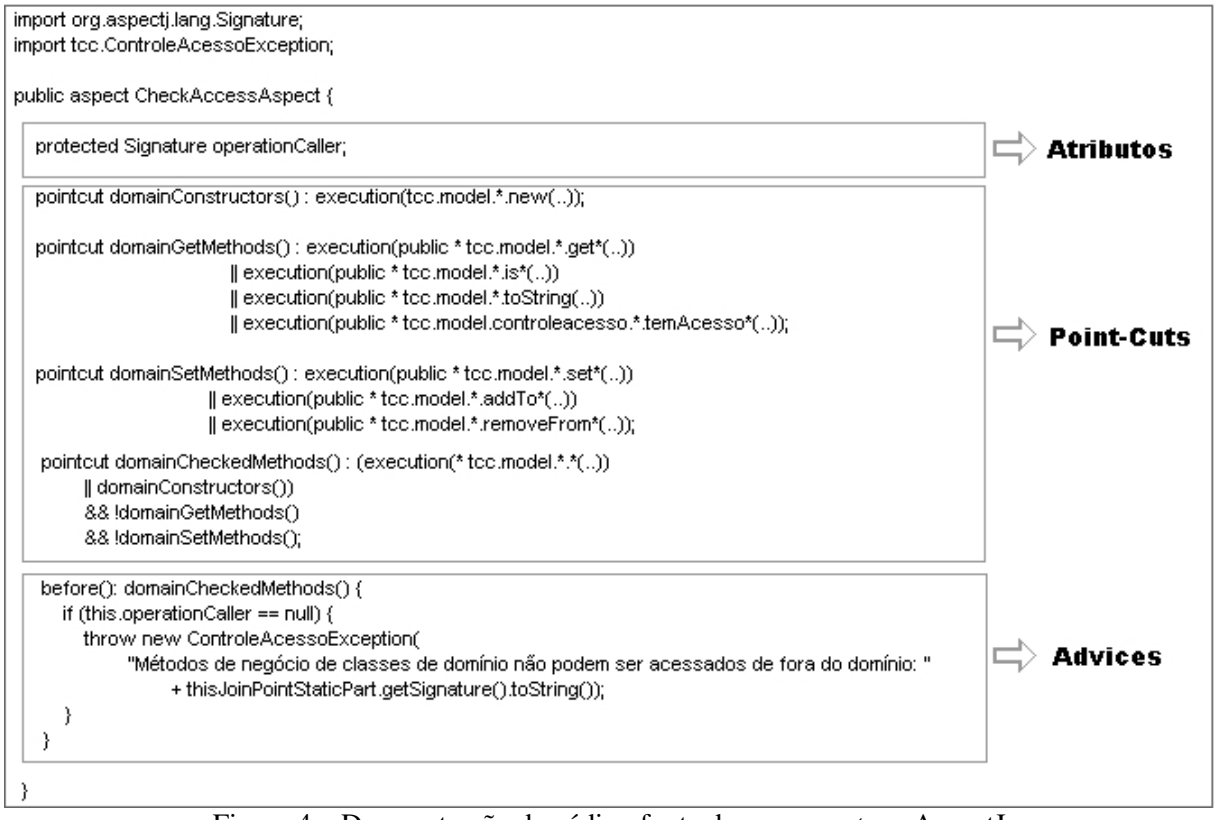


Figura 4 – Demonstração de código fonte de um *aspect* em AspectJ

2.5.1.2 Join point

Os *join points* representam pontos bem definidos no fluxo de execução de um programa onde um determinado aspecto pode ser aplicado. Estes pontos, ou *join points,* podem ser chamadas a métodos, acessos a membros de uma classe, etc. A Figura 5 apresenta a implementação de alguns *join points*.

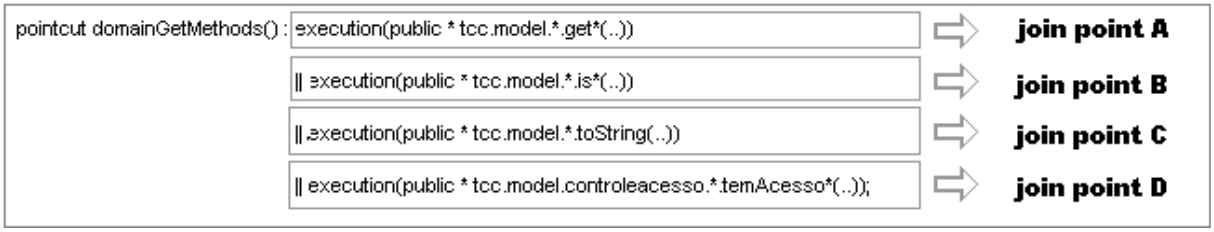


Figura 5 – Demonstração de código fonte de *join points* em AspectJ

A leitura do *join point* "A" da Figura 5 é feita da seguinte forma: Capturar a execução (***execution***) de todos os métodos públicos que possuam qualquer tipo de retorno (***public** **), de qualquer classe do pacote "tcc.model" (***tcc.model.****), que iniciem o nome com o prefixo "*get*" (***get****) e que recebam qualquer tipo e quantidade de parâmetros (***(..)***).

2.5.1.3 Point cut

O point cut do AspectJ é um agrupamento de *join points.* Um *point cut* seleciona *join points* e captura o contexto destes *join points.* Por exemplo, um *point cut* pode selecionar um *join point* que consiste na chamada de um método. O *point cut* não somente irá capturar a chamada do método através do *join point*, mas também poderá disponibilizar o contexto desta chamada como os parâmetros recebidos pelo método e o objeto no qual o método foi chamado. Para elucidar melhor a diferença entre *point cut* e *join point*, deve-se considerar um *point cut* como a definição de regras para a combinação (*weaving rules*) entre aspectos e classes, e um *join point* como situações que atendem as estas regras. A Figura 6 apresenta a

implementação de um *point cut*.

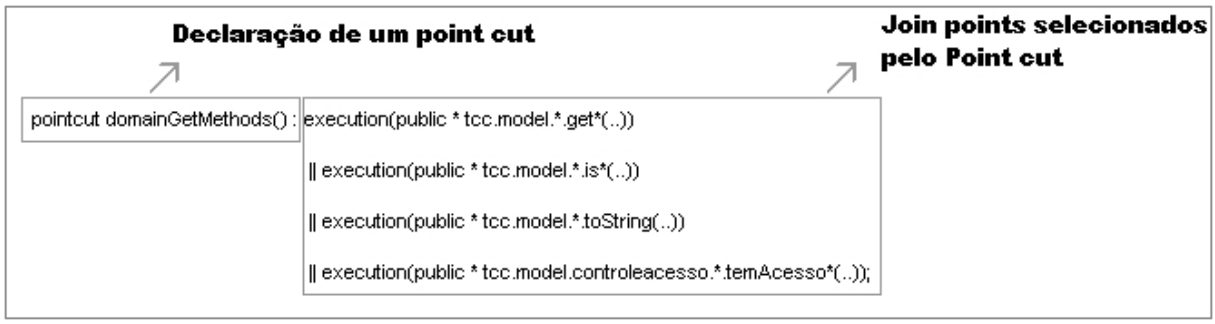


Figura 6 – Demonstração de código fonte de um *point cut* em AspectJ

2.5.1.4 Advice

Advices são trechos de código que são executados nos *join points* capturados pelos *point cuts*. Um *advice* contém as alterações que devem ser aplicadas ortogonalmente ao sistema, e podem ser executados antes (*before*), após (*after*) e por toda parte (*around*) dos *join points*. A implementação de um *advice* é muito semelhante a de um método Java. A Figura 7 apresenta a implementação de um *advice*.

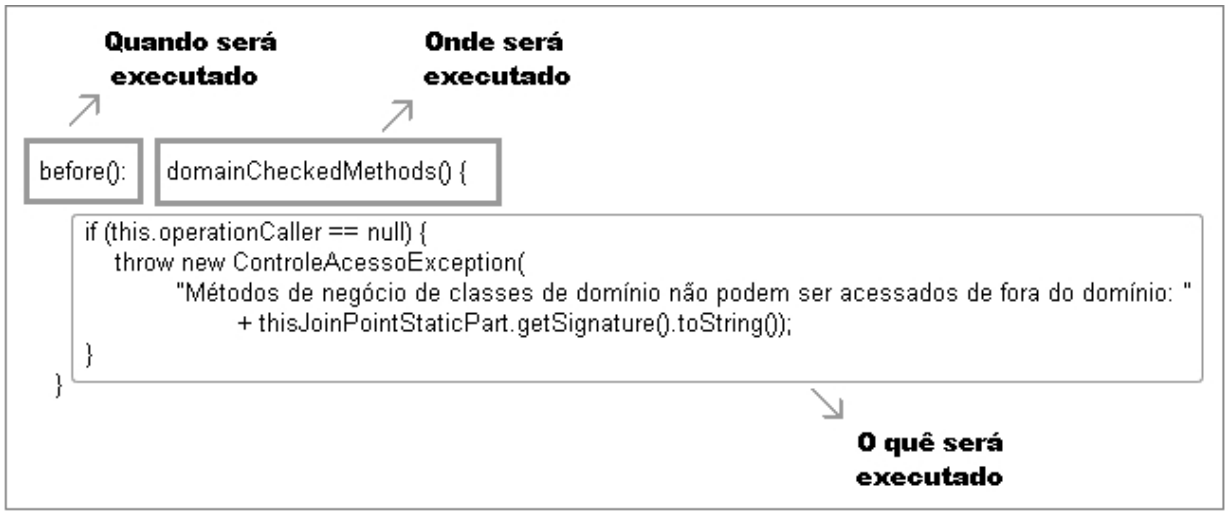


Figura 7 – Demonstração de código fonte de um *advice* em AspectJ

A leitura do *advice* apresentado na Figura 7 é feita da seguinte forma: antes (***before***) de executar os *join points* capturados pelo *pointcut* "domainCheckedMethods()", deve ser executado o seguinte código ("**o que será executado"**).

2.5.1.5 Introductions

Introductions são *crosscutings* estáticos *(static crosscuting)* que introduzem alterações à estrutura de classes, interfaces e até mesmo *aspects.* Com este mecanismo é possível adicionar membros (atributos, métodos e construtores), alterar a hierarquia das classes, e converter exceções checadas para não checadas, como, por exemplo, converter uma exceção checada do tipo "java.io.IOException" para uma exceção não checada do tipo "org.aspectj.lang.SoftException" através do mecanismo *Exception Softening,* conforme apresentado na Figura 8.

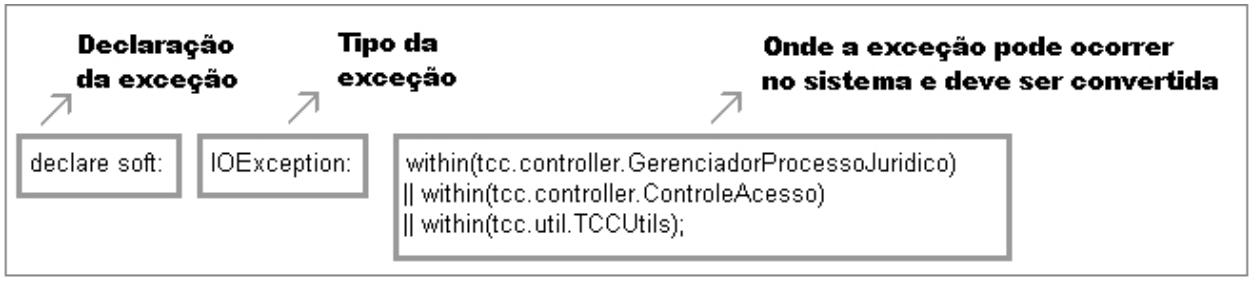


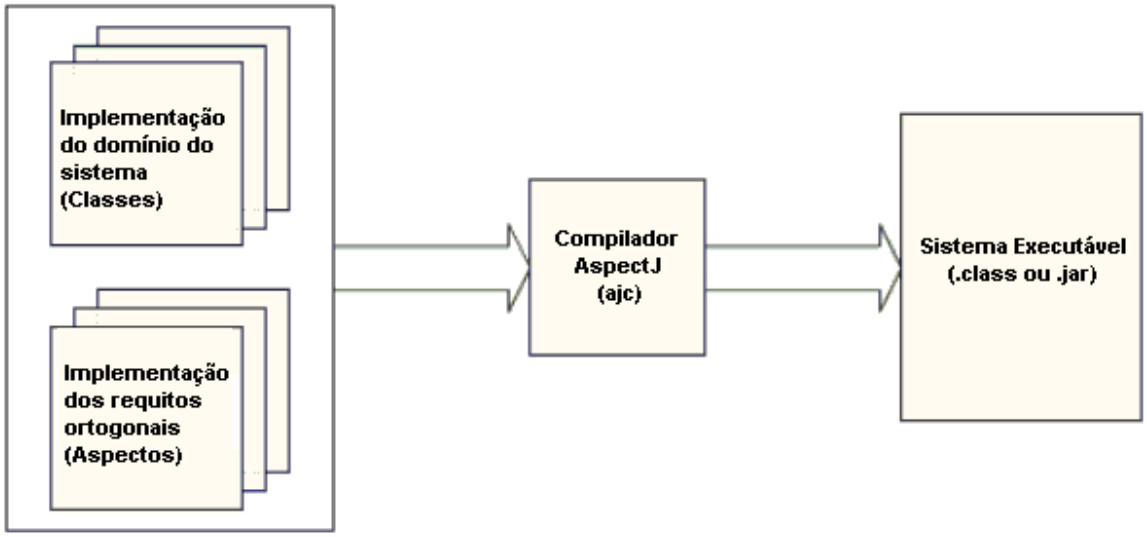
Figura 8 – Demonstração de código fonte de um *introduction* utilizando o recurso *Exception Softening* em AspectJ

A leitura da aplicação do mecanismo de *exception softening* do AspectJ, apresentado na Figura 8, pode ser feita da seguinte forma: é declarada a conversão para não checada (***declare soft***) de qualquer exceção do tipo *IOException (**IOException**)* que ocorrer nos seguintes *join points* (**"onde a exceção pode ocorrer no sistema e deve ser convertida"**).

2.5.2 Implementação da linguagem

O compilador é a parte central da implementação da linguagem AspectJ, que tem por finalidade principal combinar código de classes Java e dos Aspectos em tempo de compilação para gerar os executáveis compostos pelo código de ambos. O resultado final é *bytecode* padrão Java, e conseqüentemente poderá ser executado em qualquer JVM (*Java Virtual*

Machine). Este processo é denominado processo de combinação (*weaving).* A Figura 9 apresenta a implementação do processo de *weaving* na forma do compilador AspectJ.



Fonte: Adaptado de Laddad (2003, p. 25)

Figura 9 – Processos de combinação (*weaving*)

2.6 TRABALHOS CORRELATOS

Em Soares e Borba (2002), é relatado o desenvolvimento orientado a aspectos de aplicações utilizando a linguagem AspectJ. Neste trabalho, foi realizada uma análise comparativa entre o emprego de POO e POA no desenvolvimento de um requisito não funcional de distribuição para um sistema que registra queixas do sistema público de saúde.

O desenvolvimento orientado a aspectos também é relatado em Gradecki e Lesiecki (2003), na garantia de pré-condições relacionadas aos parâmetros recebidos por métodos Java. Os desenvolvedores geralmente precisam produzir código repetitivo para garantir que valores não sejam nulos, estejam dentro de um intervalo de valores permitido ou de acordo com uma máscara de valor pré-definida, por exemplo. Este trabalho aborda a utilização de aspectos para encapsular as verificações de pré-condições desta espécie.

Em Grott e Hugo (2005), é relatado o uso da programação orientada a aspectos em

uma aplicação B2B (*Business to business*) que consiste no envio de requisições de compra para reposição de estoque aos fornecedores, no qual se fez necessário o desenvolvimento de um mecanismo de *log*ging das operações de importação dos pedidos.

3 ESTUDO DE CASO

O sistema desenvolvido neste trabalho tem como propósito fornecer um alvo à análise comparativa e é delineado por um estudo de caso hipotético de gerenciamento de processos jurídicos. Este sistema permite ao usuário gerenciar os processos jurídicos, bem como seus andamentos e audiências.

Este estudo de caso está organizado conforme segue:

- Capítulo 4 – Desenvolvimento do trabalho – Cenário Inicial: Apresenta a composição inicial do estudo de caso e descreve o seu desenvolvimento.
- Capítulo 5 – Cenário de Solicitação de Melhoria no Sistema: Descreve a necessidade que estabelece um mecanismo de controle de acesso às funcionalidades do sistema e que deve ser incorporado ao estudo de caso inicial.
- Capítulo 6 – Desenvolvimento da segunda versão – Emprego da POO: Expõe o emprego da programação orientada a objetos na implementação do mecanismo de controle de acesso.
- Capítulo 7 – Desenvolvimento da terceira versão – Emprego da POA: Elucida o emprego da programação orientada a aspectos na implementação do controle de acesso.

Para a especificação deste estudo de caso, foi utilizada a metodologia de modelagem proposta por Wazlawick (2004), a qual se baseia em Larman (2004). Para a modelagem, foi utilizada a linguagem UML (*Unified Modeling Language*) e algumas convenções foram adotadas para manter os diagramas em um formato conciso:

a) campos obrigatórios: na descrição dos casos de uso, os campos obrigatórios podem ser identificados pelo símbolo (*).

b) exceções: as exceções representadas nos diagramas de seqüência sempre serão

retornadas ao Ator e, conseqüentemente, o fluxo principal da operação será abortado.

c) retornos: nos diagramas de seqüência, os retornos que não possuem indicação do que está sendo retornado são justamente para representar retornos vazios, operações sem retorno (*return void*).

Segundo Wazlawick (2004), o processo de desenvolvimento de software divide-se em quatro grandes fases: análise, projeto, implementação e testes. Neste trabalho, serão apresentados os principais artefatos gerados nas fases de análise, projeto e implementação.

4 ESTUDO DE CASO – DESENVOLVIMENTO DO CENÁRIO INICIAL

A primeira versão do sistema resultou na especificação e desenvolvimento do cenário inicial do estudo de caso utilizado para viabilizar a análise comparativa. O estudo de caso incide na necessidade de um suposto cliente que o levou a solicitar a uma empresa de desenvolvimento de sistemas de informação o desenvolvimento de um sistema para o gerenciamento de processos jurídicos.

4.1 ESPECIFICAÇÃO

Esta seção apresenta brevemente a concepção e a elaboração do estudo de caso e expõe os artefatos gerados na especificação do principal caso de uso existente no modelo.

4.1.1 Requisitos principais do problema a ser trabalhado

O Quadro 1 identifica os requisitos funcionais que foram efetivamente implementados.

Requisitos Funcionais	Implementados
RF01. O sistema deverá permitir o cadastro e manutenção de pessoas.	X
RF02. O sistema deverá permitir o cadastro e manutenção de advogados.	X
RF03. O sistema deverá manter os processos jurídicos.	X
RF04. O sistema deverá manter os andamentos do processo jurídico.	X
RF05. O sistema deverá manter as audiências do processo jurídico.	X
RF06. O sistema deverá permitir o cadastro e manutenção das áreas jurídicas, que irão caracterizar os processos.	X
RF07. O sistema deverá permitir a consulta de todos os processos jurídicos.	X

Quadro 1 – Requisitos funcionais

O Quadro 2 lista o requisito não funcional da primeira versão do sistema.

Requisitos Não Funcionais	Contemplados
RNF01. O sistema deve executar independente de sistema operacional.	X

Quadro 2 – Requisito não funcional.

4.1.2 Modelos de Casos de Uso

A Figura 10 apresenta o diagrama de casos de uso do estudo de caso.

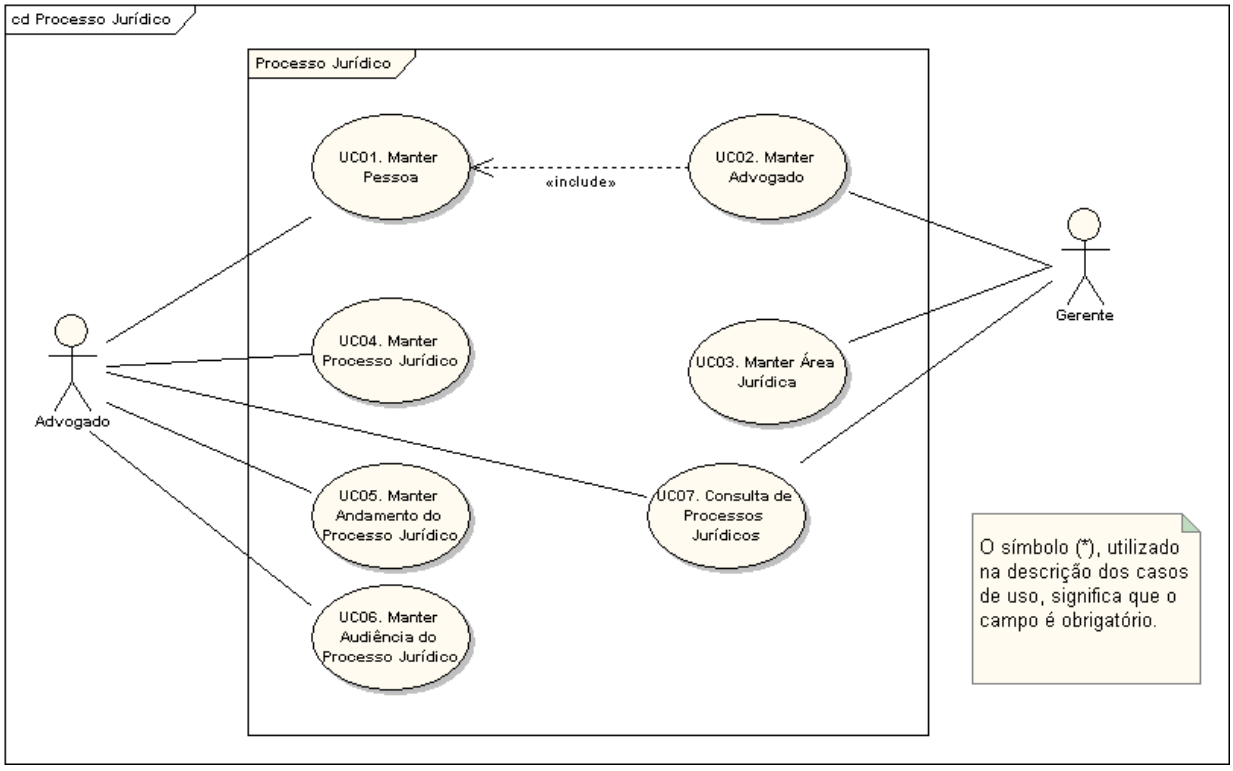


Figura 10 – Diagrama de casos de uso do sistema

4.1.3 Descrição dos Casos de Uso

No Quadro 3 está descrito apenas o principal caso de uso do sistema, “UC04. Manter Processo Jurídico”.

Sumário	Realizar abertura e manutenção do processo jurídico.
Ator	Advogado
Pré-condições	- O Cliente a ser associado ao Processo deve estar devidamente registrado. - O Advogado a ser associado ao Processo deve estar devidamente registrado. - O Réu a ser associado ao Processo deve estar devidamente registrado. - A Testemunha a ser associada ao Processo deve estar devidamente registrada. - A Área Jurídica a ser associada ao Processo deve estar devidamente registrada.
Fluxo principal (Abertura Processo Jurídico)	1. Advogado informa seu número OAB (*). 2. Advogado informa cpf do Cliente (*). 3. Advogado informa cpf do Réu (*). 4. Advogado informa cpf da Testemunha. 5. Advogado informa código da Área Jurídica (*). 6. Advogado informa Foro (*), Vara (*), Assunto (*) e Data de Abertura (*). 7. Sistema efetua a gravação, exibe o código do Processo que foi gravado e uma mensagem de sucesso.
Fluxo alternativo "Alteração dos dados de um Processo Jurídico"	1. Advogado informa código do Processo (*) 2. Sistema exibe dados do Processo. 3. Advogado altera os dados desejados do Processo. 4. Sistema efetua a gravação e exibe mensagem de sucesso.
Fluxo alternativo "Finalizar um Processo Jurídico"	1. Advogado informa código do Processo (*) 2. Sistema exibe dados do Processo. 3. Advogado informa a data do fechamento (*) e o motivo (*). 4. Sistema executa gravação e exibe mensagem de sucesso.
Fluxo de exceção	Não há.
Pós-condições	O Processo Jurídico será aberto, alterado ou finalizado.

Quadro 3 – Descrição do caso de uso "Manter Processo Jurídico"

4.1.4 Modelo Conceitual

Com o encerramento do levantamento de requisitos e da expansão dos casos de uso, foi desenvolvido o modelo conceitual. Este modelo trata-se de um artefato do domínio do problema, e não da solução tecnológica proposta. A Figura 11 apresenta o diagrama do modelo conceitual.

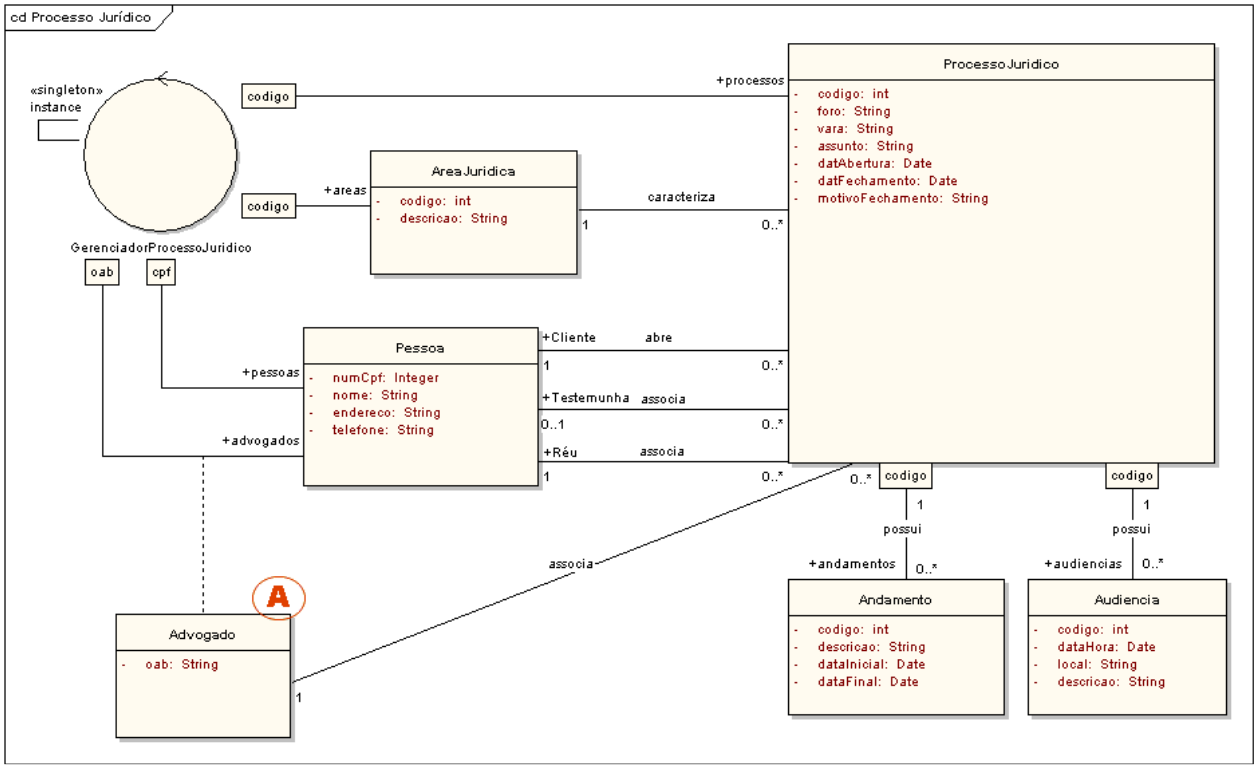


Figura 11 – Modelo conceitual do sistema

Durante o desenvolvimento deste modelo conceitual, surgiram algumas dúvidas em relação a modelagem da classe Advogado (Figura 11-A). Uma destas dúvidas é caracterizada pelo seguinte exemplo: supondo que um Advogado queira contratar o escritório onde ele trabalha para abrir um processo jurídico. Neste caso ele estaria se comportando como um Cliente, porém, não faria sentido prever neste modelo conceitual que processos jurídicos podem ser abertos por advogados e por clientes, porque o procedimento seria exatamente o mesmo. No entanto, se não houver associação entre o advogado que abriu o processo jurídico e processo em si, não é possível identificar quem teria sido o cliente daquele processo. Uma solução seria cadastrar o advogado também como cliente, deste modo o mesmo elemento ficaria com dois registros no sistema, como advogado e como pessoa. Isso gera redundância de dados e pode também gerar inconsistências, pois se a pessoa mudar de endereço, pode ser que esta mesma pessoa como advogado continue com o endereço antigo.

Cabe então questionar qual a real necessidade de representar um Advogado no modelo conceitual. Para abrir um processo jurídico, será necessário informar qual o Advogado que irá

advogar neste processo. Se fosse modelada uma classe associativa chamada “Advogado” entre “Pessoa” e “ProcessoJuridico”, a instância da classe “Advogado” apenas existiria caso ela fosse associada a uma instância de “ProcessoJuridico”. Desta forma, não existiriam advogados que ainda não trabalharam em processos jurídicos. Os advogados seriam “descobertos” após a abertura dos processos jurídicos. Esta modelagem não satisfaria o requisito “RF02“ que descreve a necessidade dos advogados estarem previamente cadastrados e tornaria a associação de um advogado a um processo jurídico não sistêmica e empírica, pois para associar um Advogado ao “ProcessoJuridico” seria necessário identificar um advogado entre as instâncias de Pessoa, ou então entre os advogados que já estivessem sido identificados e associados a algum “ProcessoJuridico”.

A modelagem realizada neste trabalho cria uma classe associativa entre “Pessoa” e o controlador-fachada “GerenciadorProcessoJuridico”, chamada “Advogado”. Esta associação é qualificada através do código único OAB, que permite identificar um advogado específico da coleção de advogados contida do controlador. Portanto, para abrir um processo jurídico, o controlador fornecerá todos os advogados cadastrados no sistema e identificados unicamente pelo seu OAB.

Não faria sentido também modelar a classe “Advogado” como subclasse da classe “Pessoa”, porque um advogado não é um tipo de pessoa diferente e sim uma relação com os processos jurídicos.

4.1.5 Modelo de Domínio

O diagrama de classes do projeto é um aprimoramento do modelo conceitual que consiste na adição dos métodos às classes, na determinação da direção das associações especificando a visibilidade entre as classes e no detalhamento dos atributos. Este diagrama

surge a partir do modelo conceitual e é enriquecido através dos modelos dinâmicos que são apresentados na seção seguinte. A Figura 12 apresenta o diagrama de classes do projeto desenhado na fase de projeto do sistema.

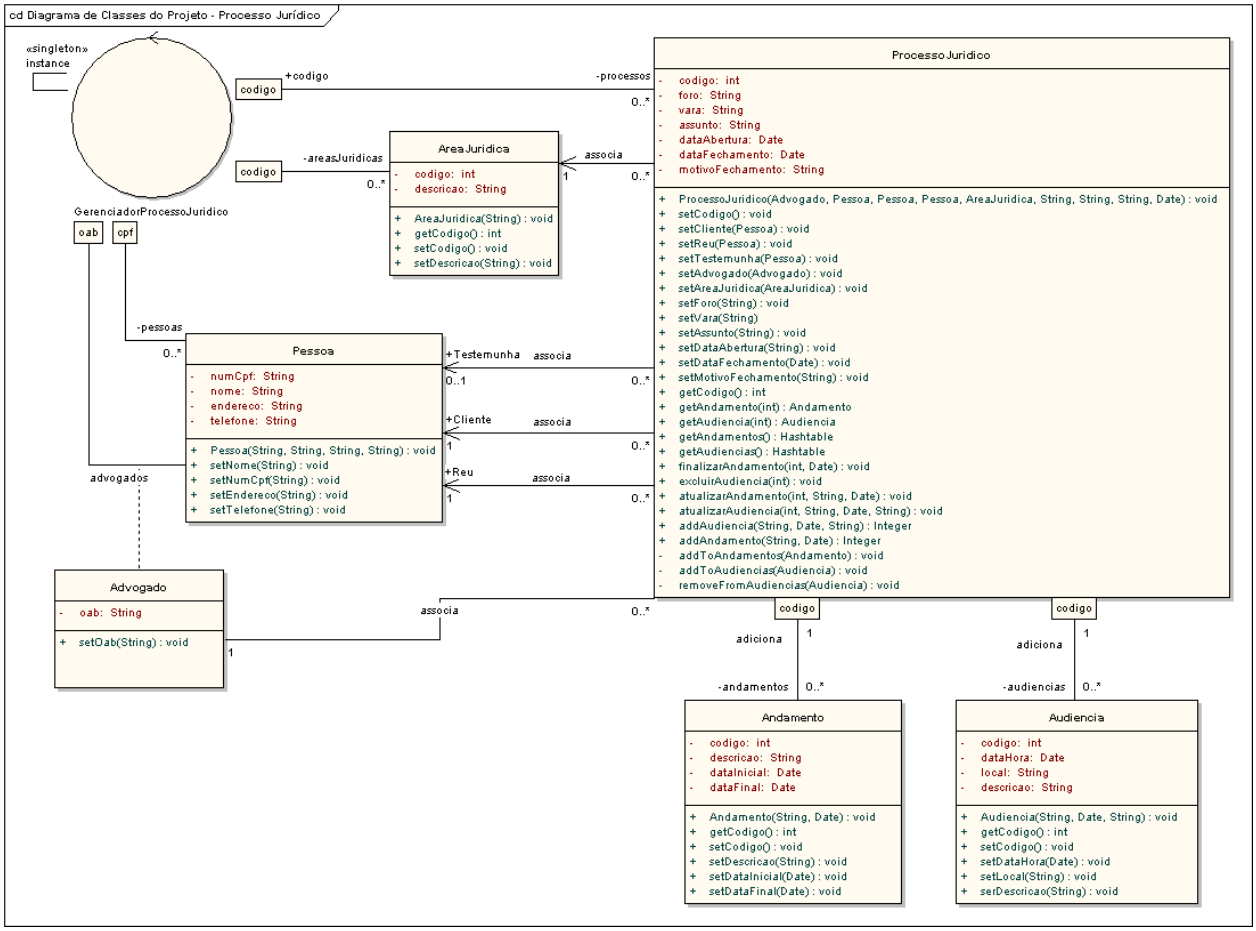


Figura 12 – Diagrama de classes do projeto

A Figura 12 demonstra o emprego do padrão de projeto s*ingleton* no controlador "GerenciadorProcessoJuridico", o que garante que haverá apenas uma instância desta classe. O desenvolvimento deste padrão de projeto foi necessário em virtude deste controlador precisar manter o estado do sistema, ou seja, todos os processos jurídicos, pessoas, advogados e áreas jurídicas registrados. Não deve existir uma instância da classe "ProcessoJuridico", por exemplo, que não esteja na coleção de processos deste controlador, mais especificamente na instância única deste controlador.

4.1.6 Diagrama de seqüência dos casos de uso

O diagrama de seqüência da UML representa a seqüência de eventos do sistema. Para cada cenário de cada caso de uso foi criado um diagrama de seqüência com o objetivo de descrever os eventos do sistema e identificar as operações de sistema.

As Figuras 13, 14 e 15 apresentam a seqüência de eventos para cada cenário do caso de uso “UC04. Manter Processo Jurídico”.

A Figura 13 descreve os eventos entre o ator, o sistema e o controlador-fachada do fluxo principal “Abertura Processo Jurídico”.

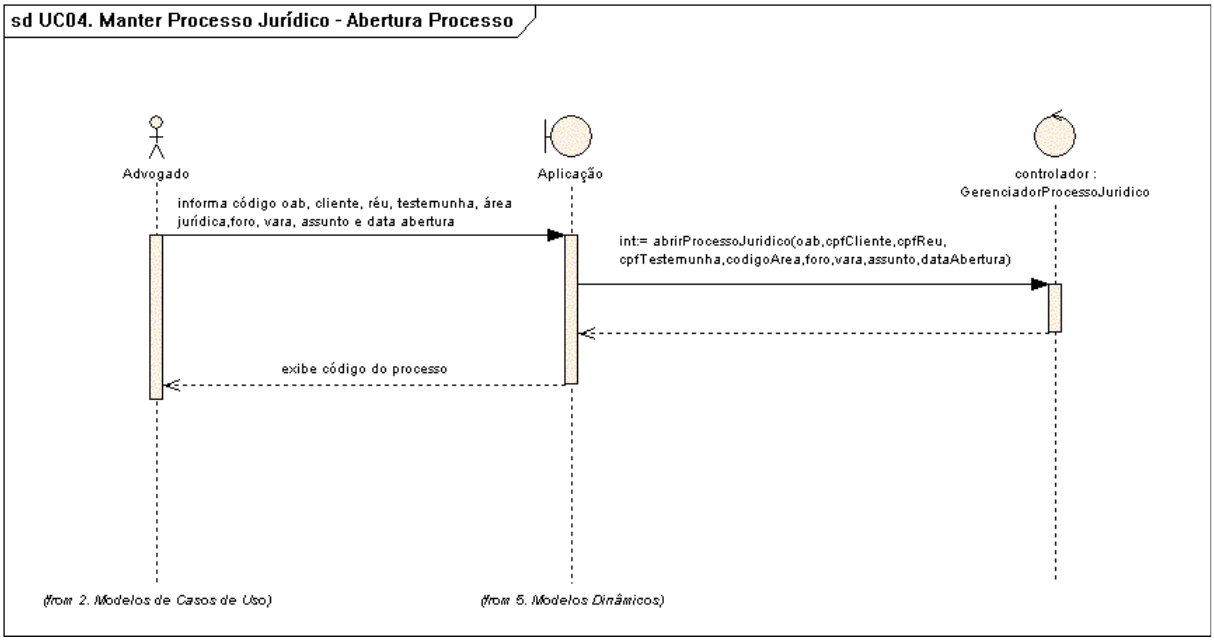


Figura 13 – Diagrama de seqüência do fluxo principal – “Abertura Processo Jurídico”

A Figura 14 descreve os eventos entre o ator, o sistema e o controlador-fachada do fluxo alternativo “Alteração dos dados de um Processo Jurídico”.

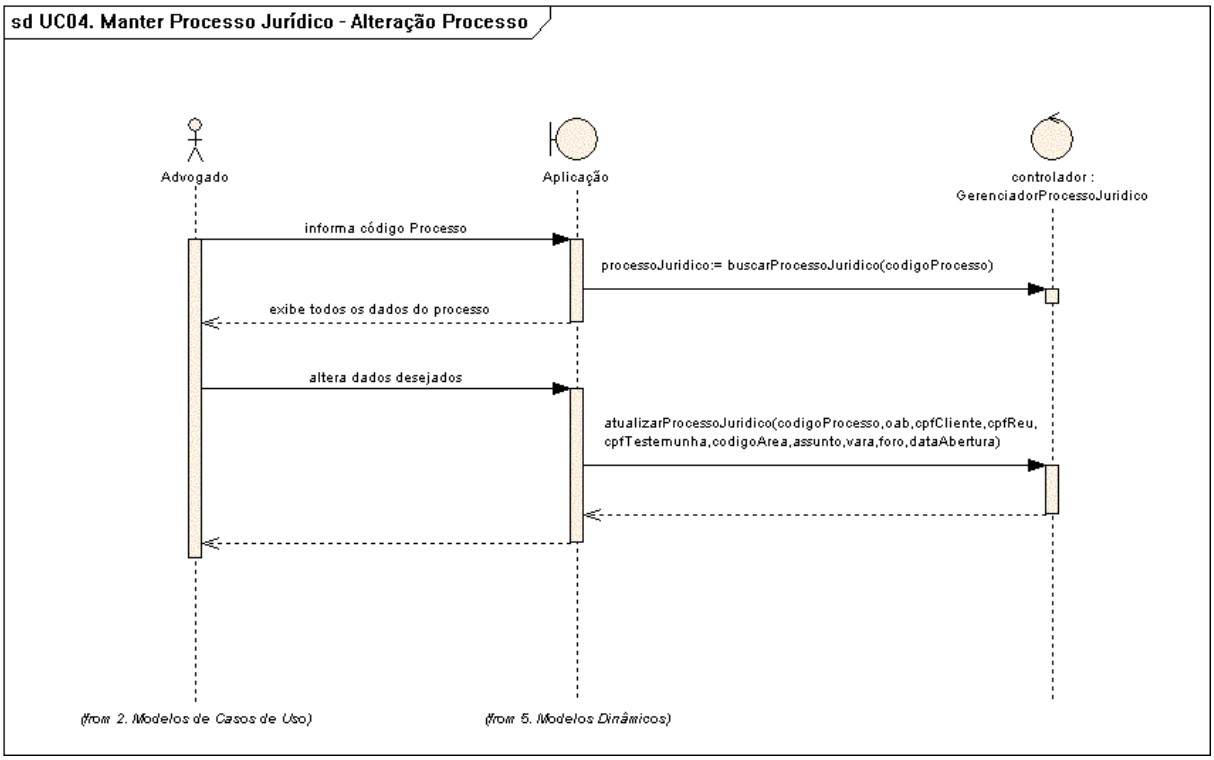


Figura 14 – Diagrama de seqüência do fluxo alternativo – "Alteração dos dados de um Processo Jurídico"

A Figura 15 descreve os eventos entre o ator, o sistema e o controlador-fachada do fluxo alternativo "Finalizar um Processo Jurídico".

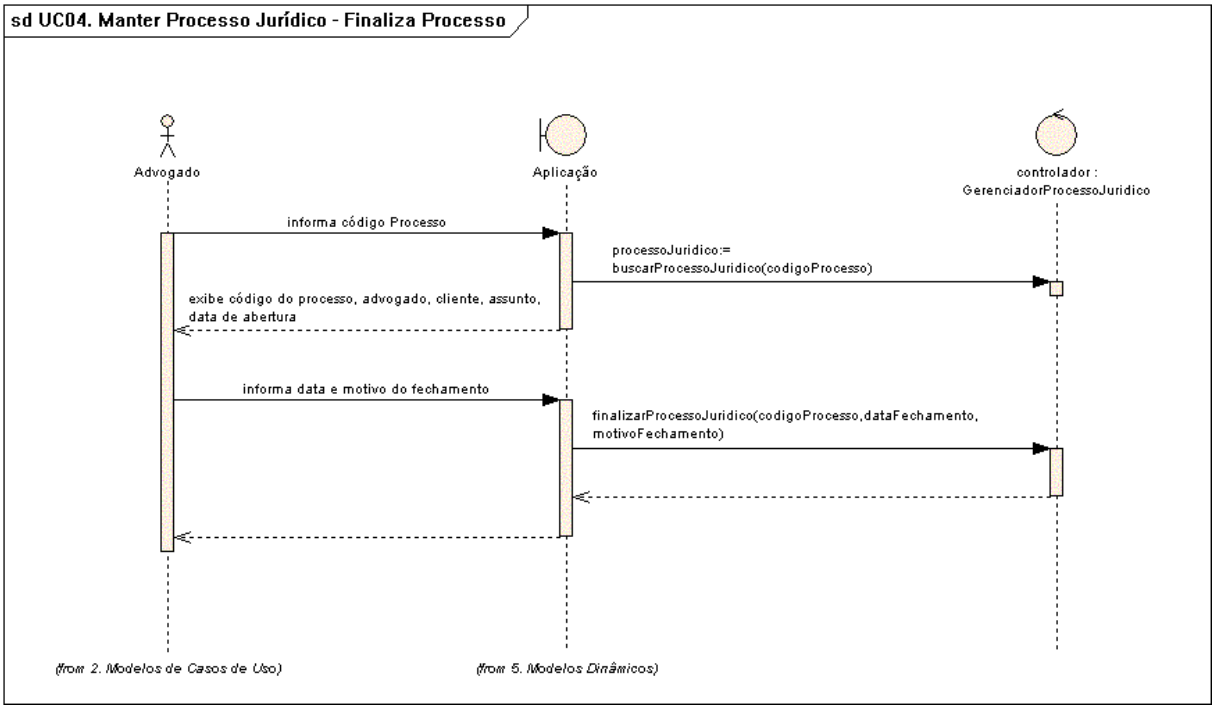


Figura 15 – Diagrama de seqüência do fluxo alternativo – "Finalizar um Processo Jurídico"

4.1.7 Diagrama de seqüência das operações do sistema

Uma vez identificadas as operações do sistema através do diagrama de seqüência dos casos de uso, são desenvolvidos os diagramas de seqüência para cada operação do sistema.

Os diagramas de seqüência das operações têm por objetivo demonstrar a interação entre as instâncias das classes, através da troca de mensagens, na descoberta dos métodos das classes de domínio.

As Figuras 16, 17 e 18 apresentam a interação das operações “abrirProcessoJuridico”, “atualizarProcessoJuridico” e “finalizarProcessoJuridico”.

A Figura 16 descreve a seqüência entre o sistema, a instância do controlador-fachada e a criação da instância “novoProcessoJuridico” para a operação de sistema “abrirProcessoJuridico”.

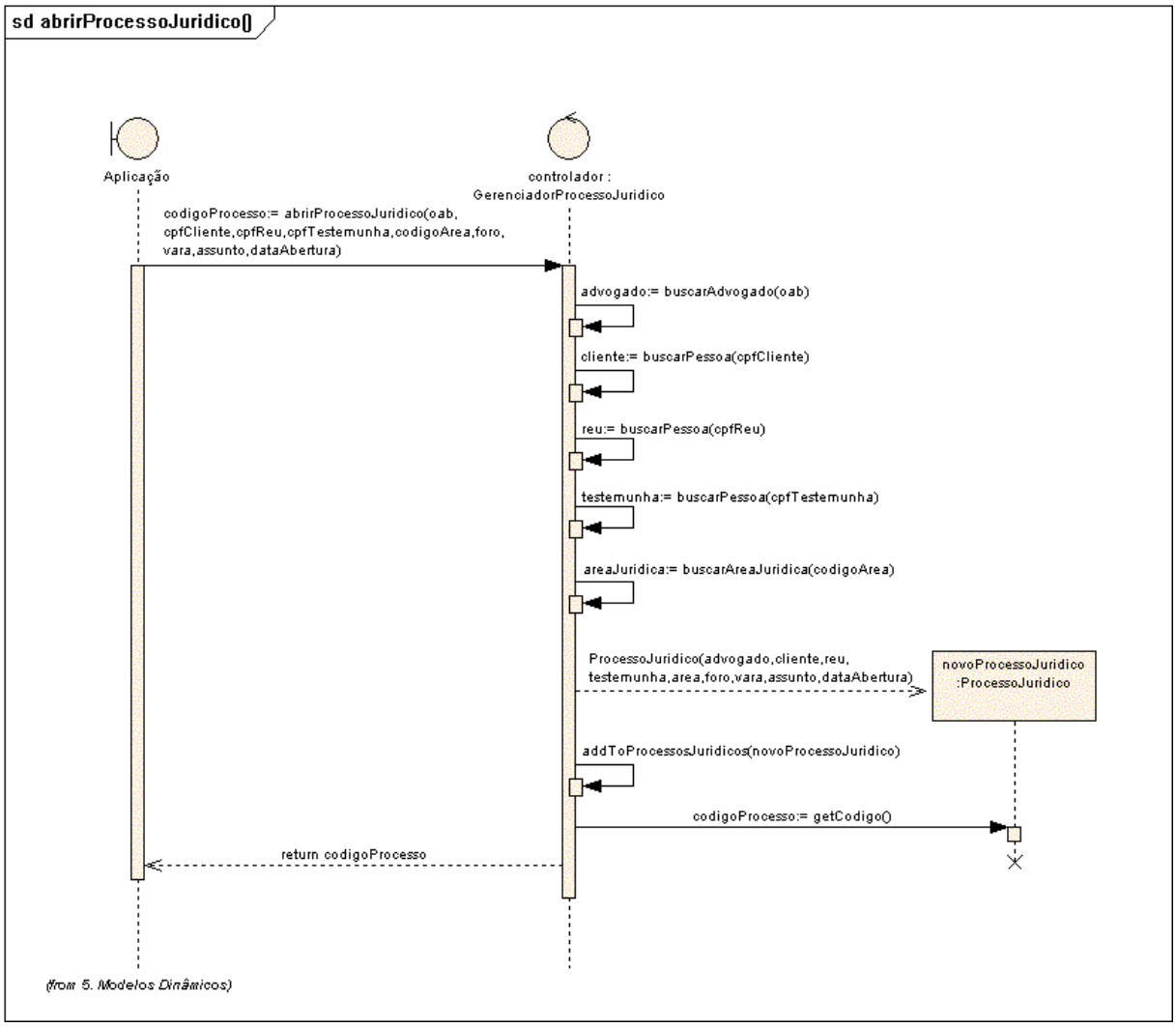


Figura 16 – Diagrama de seqüência da operação “abrirProcessoJuridico”.

A Figura 17 descreve a interação entre o sistema, a instância do controlador-fachada e a alteração do estado da instância “processoJuridico” para a operação de sistema “atualizarProcessoJuridico”.

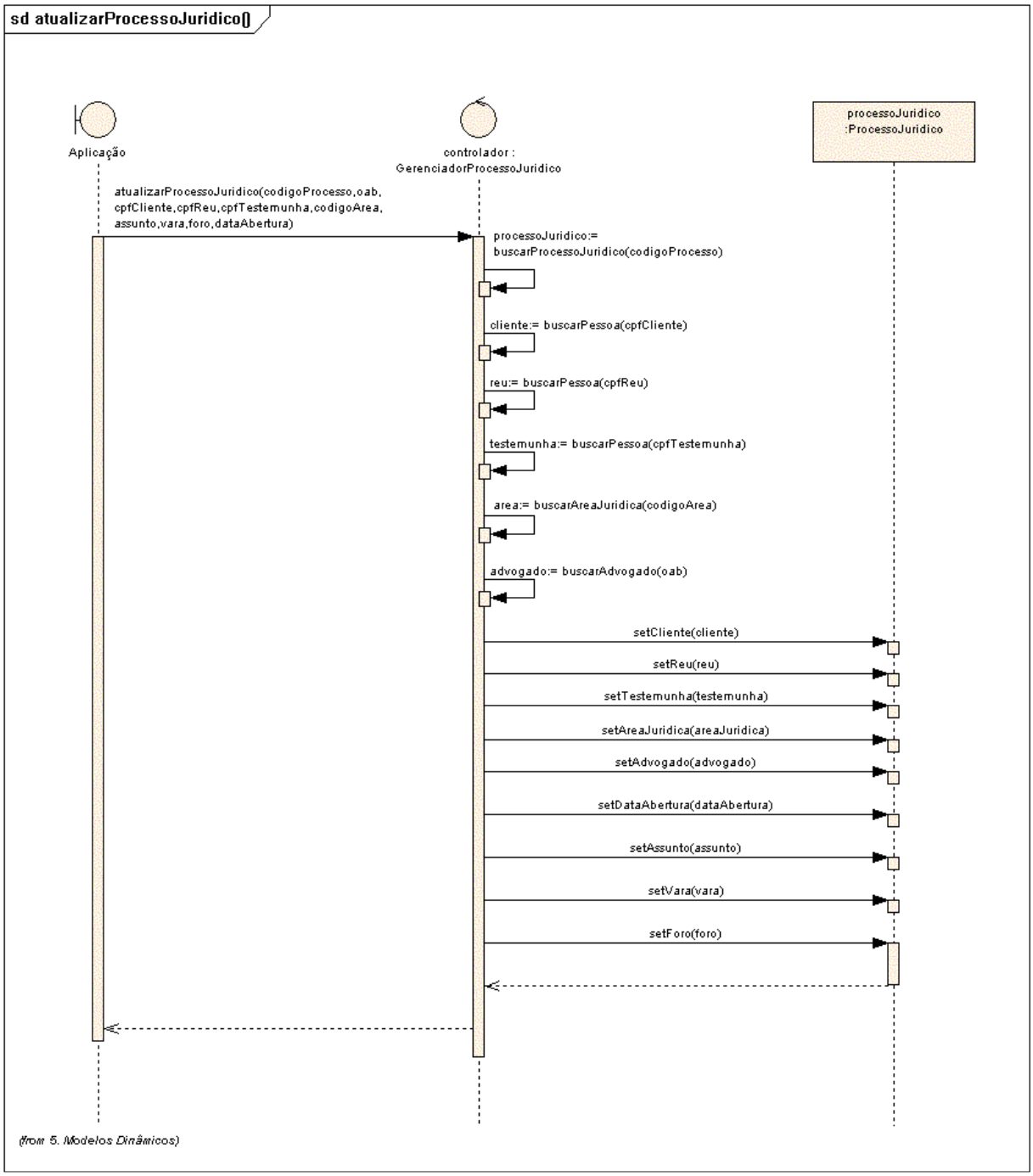


Figura 17 – Diagrama de seqüência da operação “atualizarProcessoJuridico”

A Figura 18 descreve a interação entre o sistema, a instância do controlador-fachada e a alteração do estado da instância “processoJuridico” para a operação de sistema “finalizarProcessoJuridico”.

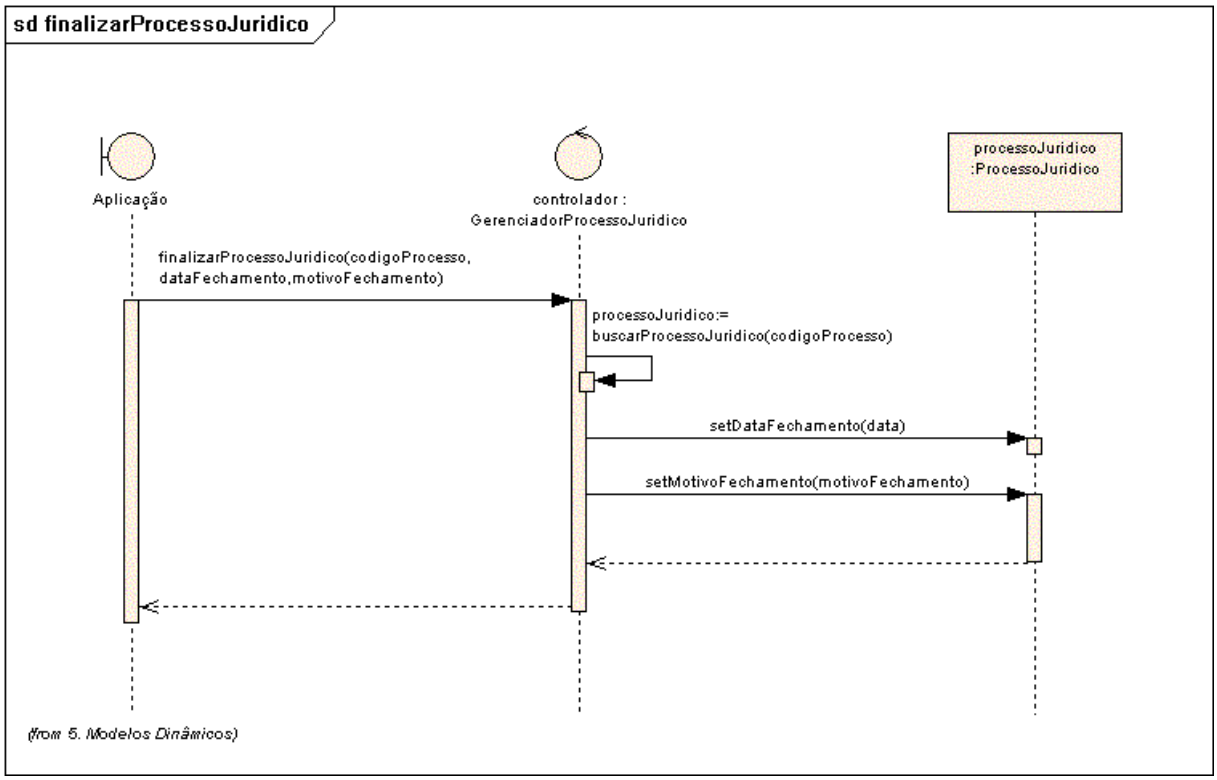


Figura 18 – Diagrama de seqüência da operação “finalizarProcessoJuridico”.

4.2 CONSIDERAÇÕES FINAIS

Neste ponto do trabalho, o sistema implementa as funcionalidades requeridas no início do projeto(pág. 39). O sistema então foi desenvolvido, conforme especificado neste capítulo, e implantado no escritório do cliente. No próximo capítulo será contextualizado um novo cenário em que o cliente solicita uma melhoria no sistema.

5 CENÁRIO DE SOLICITAÇÃO DE MELHORIA NO SISTEMA

Durante a suposta prática deste sistema, o cliente identificou a necessidade de um mecanismo de controle de acesso que deve ser provido pelo sistema e que não havia sido previsto na primeira versão do mesmo (nem pelo cliente, nem pelo analista do sistema).

Conseqüentemente, houve a solicitação de um novo requisito não funcional que deverá ser aplicado à versão original, gerando assim uma segunda versão do sistema.

5.1 ESPECIFICAÇÃO

Os requisitos funcionais do sistema permanecem os mesmos da primeira versão. Dois novos requisitos não funcionais foram adicionados em virtude do levantamento das novas necessidades do cliente. Os dois novos requisitos, “RNF02” e “RNF03” estão descritos no Quadro 4.

Requisitos Não Funcionais	Contemplados
RNF01. O sistema deve executar independente de sistema operacional.	X
RNF02. O sistema deverá prover o mecanismo de controle de acesso às funcionalidades do sistema. Os perfis de acesso serão definidos através de associações entre usuários, operações e telas do sistema.	X
RNF03. O Controle de acesso deverá ser implementado nas operações de sistema e nas telas do sistema.	X

Quadro 4 – Requisitos não funcionais

O requisito “RNF02” descreve a necessidade do mecanismo de autorização ao acesso das funcionalidades do sistema. O requisito “RNF03” descreve a necessidade da implementação deste requisito em dois níveis diferentes, operações de sistema e telas do sistema, o que causa modificações nas três camadas do sistema original: modelo, controle e visão.

A necessidade da contemplação do requisito “RNF03” se deve ao fato de que camada de modelo possivelmente será disponibilizada para o acesso de outros sistemas. Deste modo, o controle de acesso será especificado e desenvolvido para que as camadas de visão, modelo e controle satisfaçam o requisito de forma independente. Ou seja, as camadas deverão satisfazer o requisito de tal forma que a autorização seja garantida mesmo que a camada de controle e modelo não forem acessadas pelas telas do sistema, e que os usuários do sistema original apenas poderão acessar as telas se estiverem devidamente autorizados.

Para realizar a liberação de privilégios que possibilite aos usuários acessarem as telas do sistema, bem como possibilitar a outros sistemas acessarem as operações do gerenciador de processos jurídicos, foi especificado um módulo de controle de acesso para o sistema, que permite o cadastro de usuários, cadastro de telas, cadastro de operações de sistema, liberação e remoção de privilégios. Este módulo servirá como ferramenta para a administração dos usuários e privilégios do sistema, disponibilizando os dados necessários ao controle de acesso das funcionalidades do sistema.

A Figura 19 apresenta os casos de uso do módulo controle de acesso.

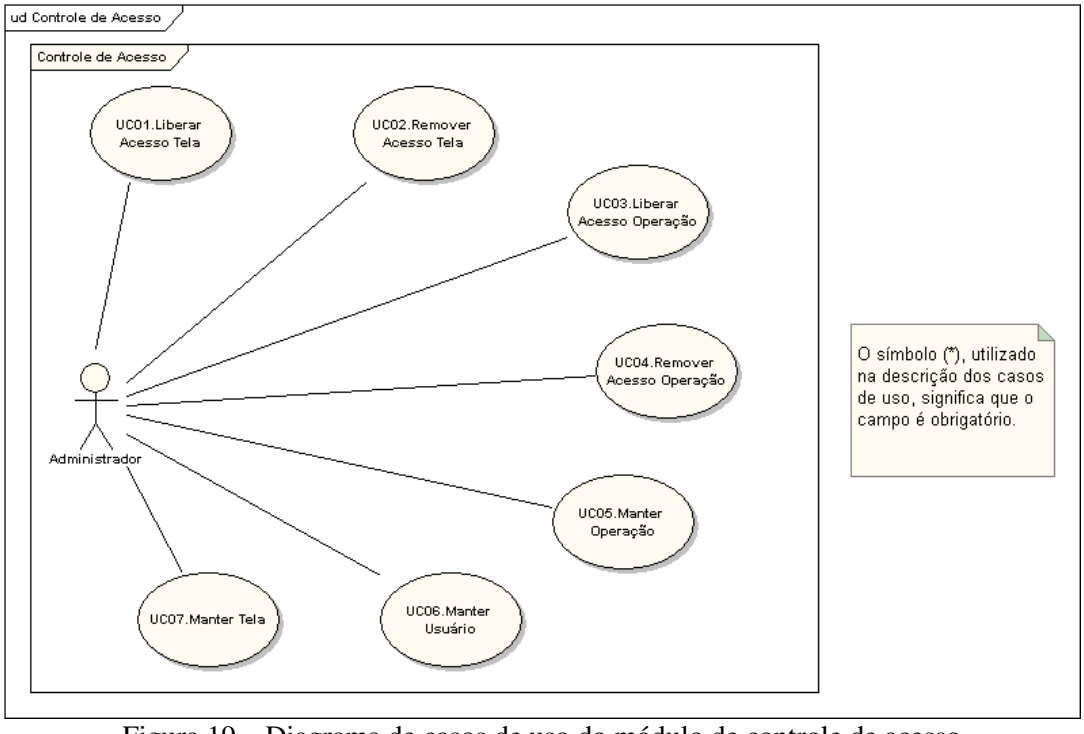


Figura 19 – Diagrama de casos de uso do módulo de controle de acesso

No contexto deste trabalho, apenas os casos de uso “UC01. Liberar Acesso Tela”, “UC02. Remover Acesso Tela” e “UC06. Manter Usuário” foram implementados.

Julgou-se desnecessária a implementação de todos os casos de uso deste módulo em relação ao objetivo principal do trabalho. As operações de sistema e as telas foram previamente cadastradas em uma base que é carregada ao iniciar a aplicação, através de um método que executa uma carga inicial de dados, criando os objetos que armazenam as telas e operações do sistema.

5.1.1 Modelo conceitual do módulo de controle de acesso

Para elucidar o modelo utilizado no módulo de controle de acesso, é apresentado na Figura 20 o modelo conceitual deste módulo. Este modelo contém, além das classes de domínio da aplicação, um controlador-fachada que disponibiliza as operações de sistema para o gerenciamento das autorizações e os métodos necessários do controle de acesso ao sistema principal.

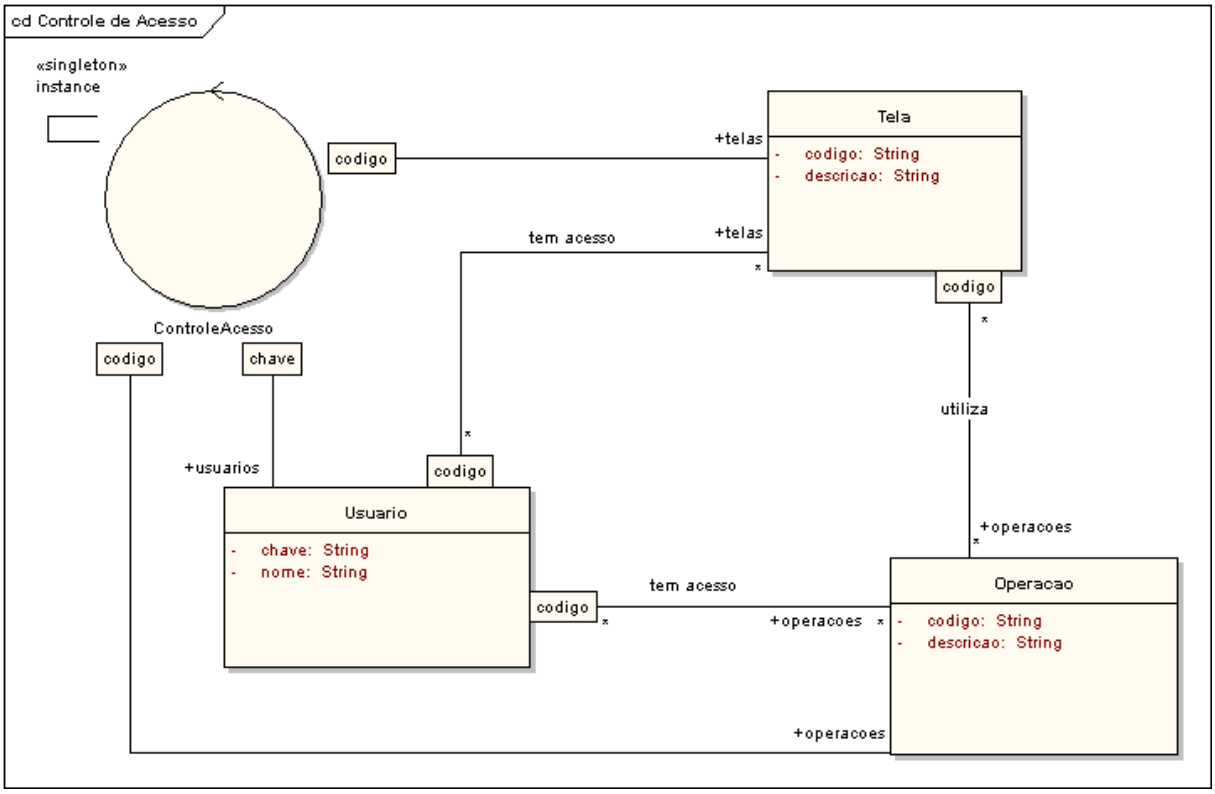


Figura 20 – Modelo conceitual do módulo controle de acesso

O próximo capítulo descreve o projeto e a implementação da segunda versão do sistema.

6 DESENVOLVIMENTO DA SEGUNDA VERSÃO – EMPREGO DA POO

A segunda versão do sistema agrega os novos requisitos não funcionais de controle de acesso que devem ser contemplados pela aplicação original, utilizando somente programação orientada a objetos.

Estes requisitos foram confrontados com os requisitos funcionais da versão original do sistema. O Quadro 5 apresenta a matriz de relacionamento entre os requisitos original e os novos requisitos.

Requisitos Funcionais do Sistema Original	**Novos Requisitos não Funcionais**	
	RNF02.	RNF03
RF01. O sistema deverá permitir o cadastro e manutenção de pessoas.	**X**	**X**
RF02. O sistema deverá permitir o cadastro e manutenção de advogados.	**X**	**X**
RF03. O sistema deverá manter os processos jurídicos.	**X**	**X**
RF04. O sistema deverá manter os andamentos do processo jurídico.	**X**	**X**
RF05. O sistema deverá manter as audiências do processo jurídico.	**X**	**X**
RF06. O sistema deverá permitir o cadastro e manutenção das áreas jurídicas, que irão caracterizar os processos.	**X**	**X**
RF07. O sistema deverá permitir a consulta de todos os processos jurídicos.	**X**	**X**

Quadro 5 – Matriz de relacionamento entre os requisitos funcionais e não funcionais

Como se pode observar, o impacto dos novos requisitos não funcionais afeta todos os outros requisitos funcionais que foram originalmente especificados e implementados, sem mesmo ter uma forte relação ao domínio destes requisitos. Com base no impacto que eles geram no sistema, teve-se por conhecimento que eles são fortemente caracterizados por uma natureza ortogonal e, portanto, são considerados requisitos ortogonais (*crosscuting concerns*).

6.1 ESPECIFICAÇÃO

Para representar o impacto dos novos requisitos não funcionais na segunda versão do sistema, a qual o controle de acesso foi implementado utilizando a programação orientada a objetos, é apresentada nesta seção a visão dinâmica do sistema através da modificação dos diagramas de seqüência dos casos de uso e das operações.

O controle de acesso consiste em garantir as seguintes condições:

a) Os acessos aos métodos de domínio do sistema só poderão ocorrer através do controlador-fachada principal (GerenciadorProcessoJuridico). Nenhum objeto de domínio poderá ser instanciado, ou ter ser estado alterado por outra classe que não o controlador-fachada principal.
b) Apenas usuários autorizados poderão acessar as operações de sistema através do controlador.
c) Apenas usuários autorizados poderão acessar as telas do sistema.

Para satisfazer estas condições impostas pelos novos requisitos não funcionais, o controle de acesso foi implementado em dois níveis: acesso às operações de sistema e acesso às telas do sistema. As sub-seções seguintes delineiam os impactos desta manutenção em nível de projeto do sistema original.

6.1.1 Diagrama de seqüência dos casos de uso

Os diagramas de seqüência dos casos de uso da primeira versão do sistema foram modificados para representarem também os eventos relacionados ao controle de acesso.

As Figuras 21, 22 e 23 apresentam a seqüência dos eventos para cada cenário do caso de uso “UC04. Manter Processo Jurídico”, inclusive os eventos necessários ao controle de

acesso.

A Figura 21 descreve os eventos entre o ator, o sistema, o controlador principal e o controlador do controle de acesso do fluxo principal “Abertura Processo Jurídico”. Em vermelho estão destacados os eventos invasivos do controle de acesso.

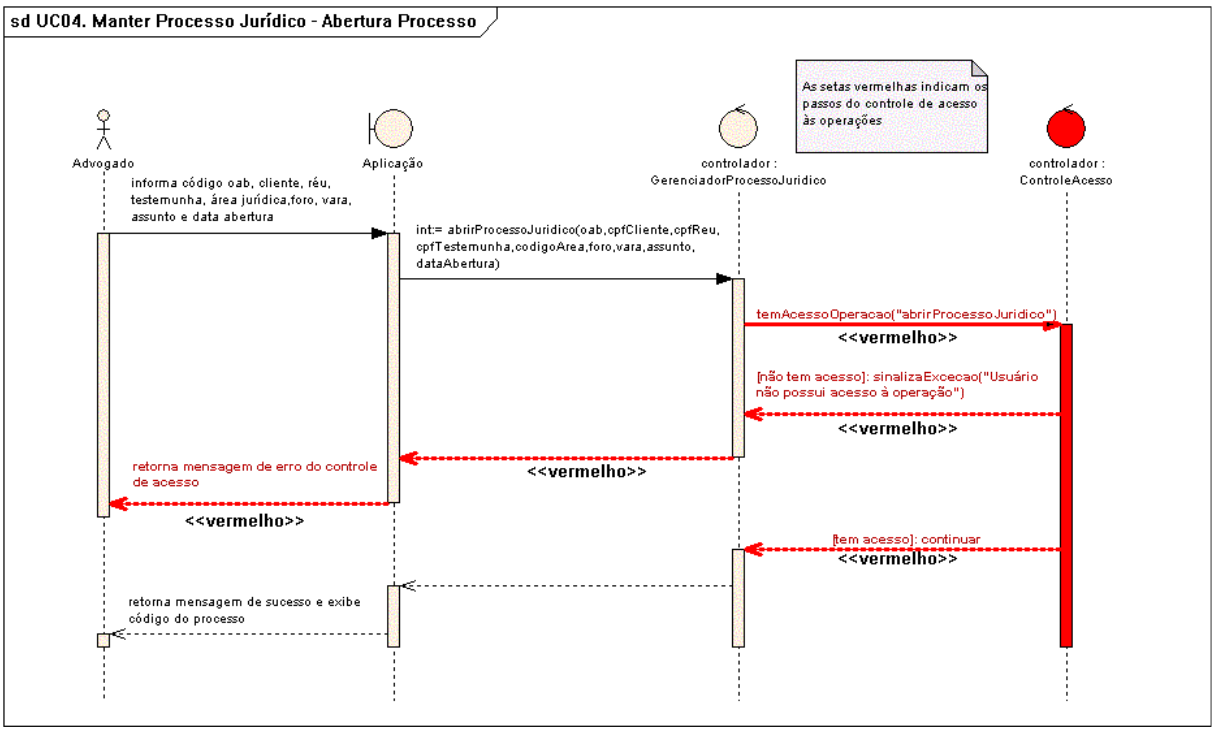


Figura 21 – Diagrama de seqüência do fluxo principal – “Abertura Processo Jurídico”

Na Figura 21 é caracterizado o fenômeno *crosscuting* que ocorre no impacto do requisito de controle de acesso sob a operação de sistema “abrirProcessoJuridico”. Esta operação tem por obrigação realizar uma ação de domínio, que é abrir um processo jurídico.

Neste momento, também é delegada a responsabilidade de garantir que o usuário autenticado no sistema esteja autorizado a executá-la. Desta forma, a lógica de domínio se mistura a um requisito não funcional de software, isto diminui a coesão porque as responsabilidades desta operação não estão fortemente relacionadas e, conseqüentemente, aumenta o acoplamento, fazendo com que este objeto (controlador: GerenciadorProcessoJuridico) tenha conhecimento de outros objetos (controlador: ControleAcesso) para poder executar uma operação.

A Figura 22 descreve os eventos entre o ator, o sistema, o controlador principal e o controlador do controle de acesso do fluxo alternativo “Alteração dos dados de um Processo Jurídico”. Em vermelho estão destacados os eventos invasivos do controle de acesso.

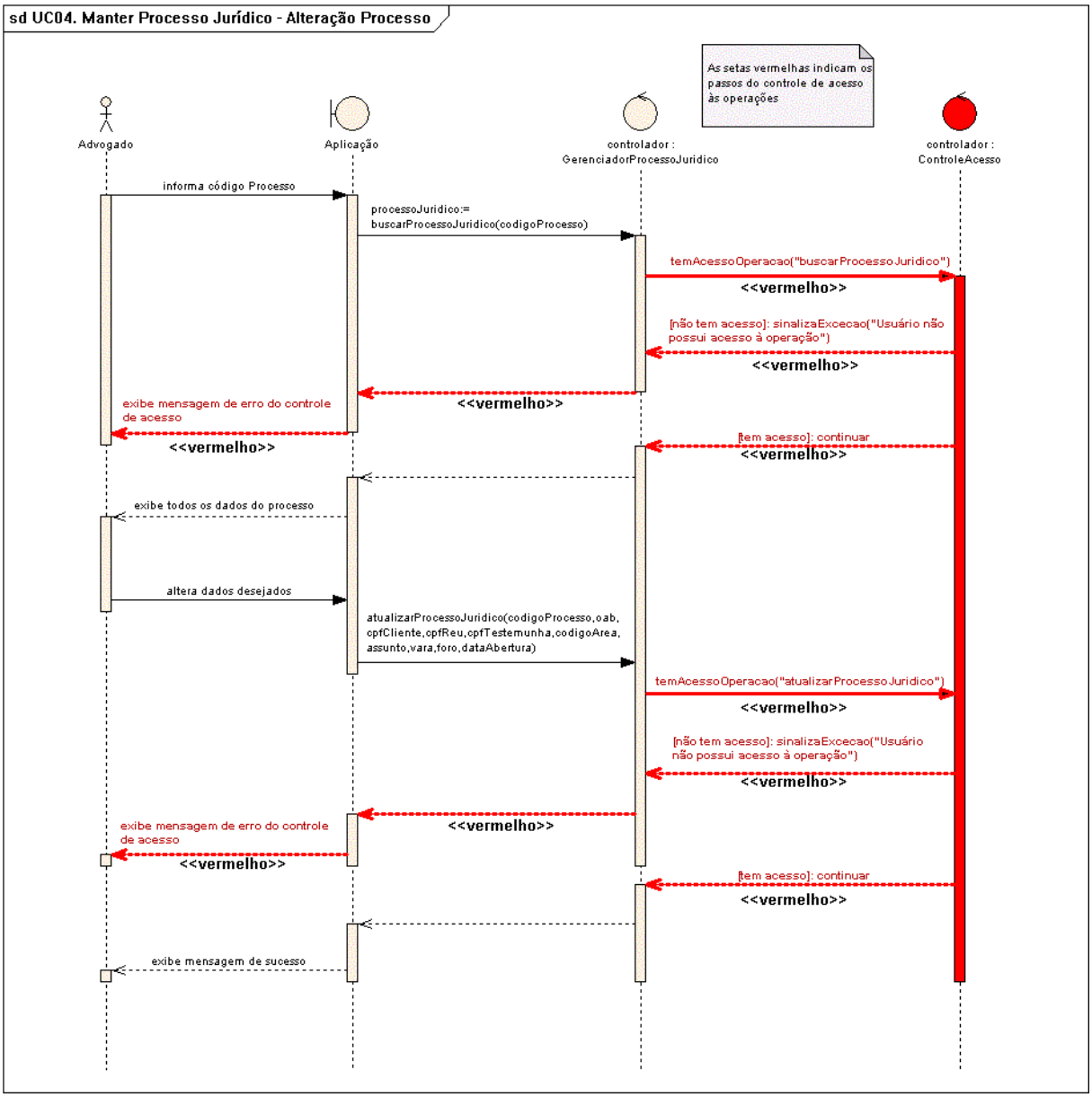


Figura 22 – Diagrama de seqüência do fluxo alternativo – “Alteração dos dados de um Processo Jurídico”

A Figura 22 apresenta o impacto do requisito ortogonal nas operações “buscarProcessoJuridico” e “atualizarProcessoJuridico”, que precisam enviar uma mensagem a um outro controlador para verificar se o usuário autenticado está autorizado a executar estas

operações.

A Figura 23 descreve os eventos entre o ator, o sistema, o controlador principal e o controlador do controle de acesso do fluxo alternativo “Finalizar um Processo Jurídico”. Em vermelho estão destacados os eventos invasivos do controle de acesso.

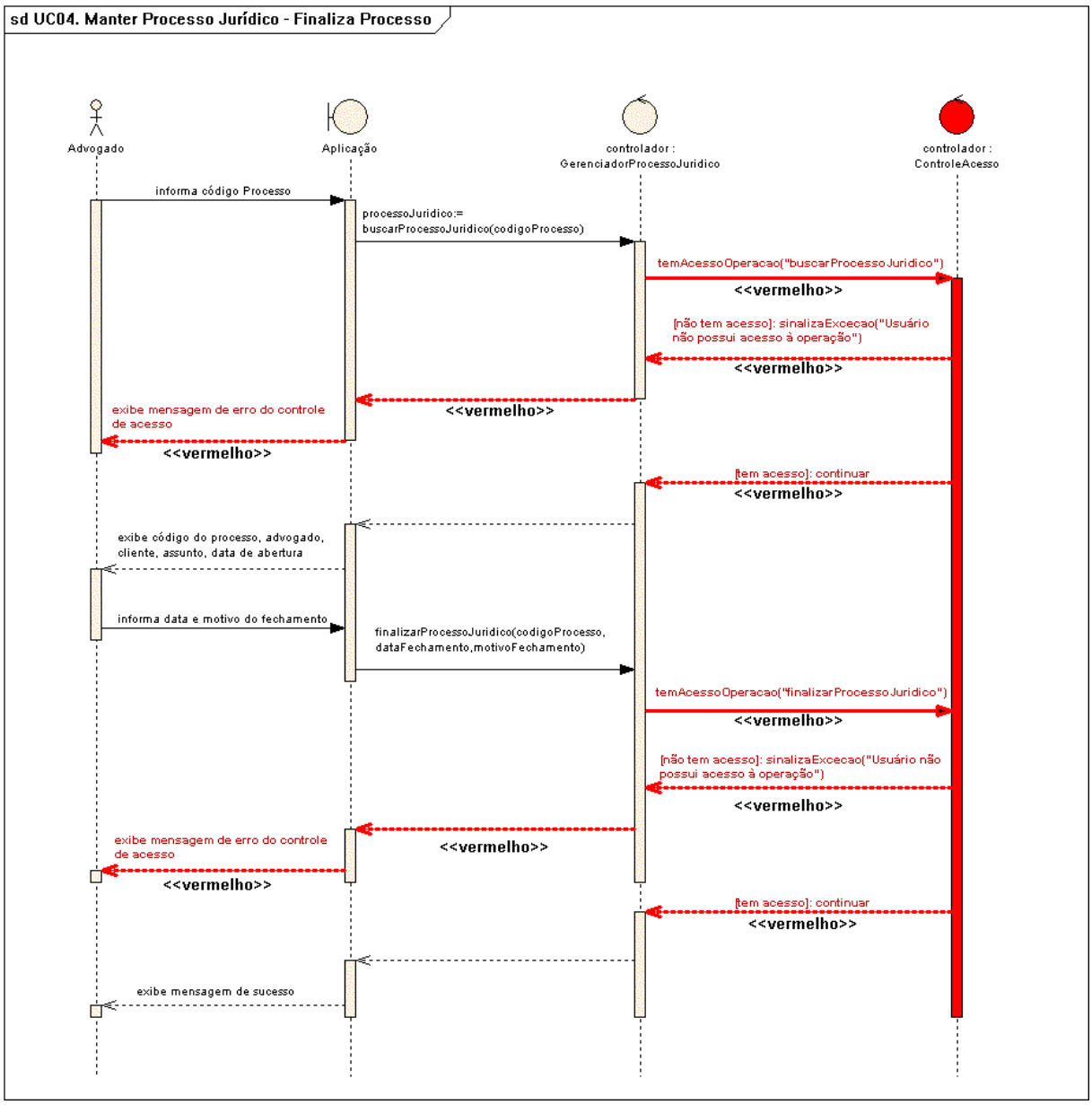


Figura 23 – Diagrama de seqüência do fluxo alternativo – “Finalizar um Processo Jurídico”

Na Figura 23, os eventos invasivos estão caracterizados pelo envio de mensagens das operações do controlador “GerenciadorProcessoJuridico” ao controlador “ControleAcesso” para satisfazer o requisito não funcional de autorização. O controlador

“GerenciadorProcessoJuridico” é um controlador-fachada do padrão GRASP. Se um controlador desta espécie executa muitas tarefas necessárias para atender ao evento do sistema, ele pode ser classificado como um “controlador inchado”, conforme Larman (2004, p. 253).

O cenário descrito na Figura 23 apenas contempla um requisito não funcional de interesse ortogonal, mas esta realidade poderia ser diferente e o controlador poderia executar várias tarefas para realizar uma única delegação a camada de domínio.

6.1.2 Diagrama de seqüência das operações do sistema

Os diagramas de seqüência das operações de sistema da primeira versão foram modificados para representarem também os passos relacionados ao controle de acesso.

A Figura 24 descreve a interação entre o sistema, o controlador principal, o controlador do controle de acesso a instância “novoProcessoJuridico” da operação de sistema “abrirProcessoJuridico” , inclusive a troca de mensagens necessária para garantir o controle de acesso às operações de sistema e ao domínio do sistema.

Os métodos de acesso (*set* e *get*) não foram alterados em virtude do controle de acesso ter a função de garantir a segurança das operações que tratam do domínio do negócio, e não os métodos básicos de acesso a atributos das classes.

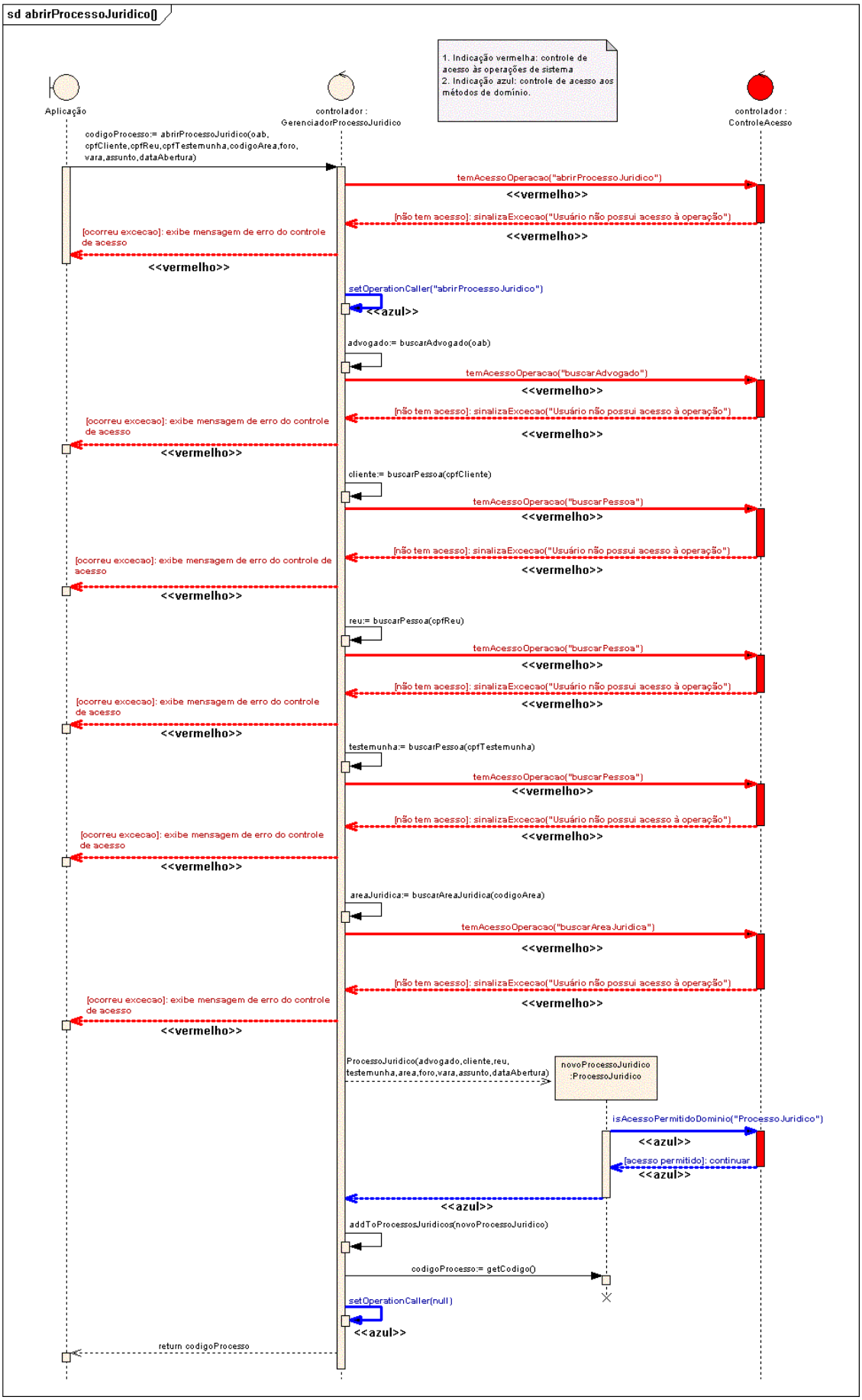


Figura 24 – Diagrama de seqüência da operação de sistema “abrirProcessoJuridico”.

Na Figura 24 também é destacado o fenômeno *crosscuting*, agora em um nível mais detalhado da operação “abrirProcessoJuridico”. Nesta operação, o controlador fachada do sistema (GerenciadorProcessoJuridico) precisa executar diversos passos para abrir um processo jurídico.

No primeira interação em azul, o controlador envia uma mensagem recursiva que irá atribuir um valor ao atributo privado “operationCaller”. Este atributo é utilizado, na segunda interação em azul, pelo controlador “ControleAcesso” para verificar se existe algum valor, o que sinaliza que a instância “novoProcessoJuridico” está efetivamente sendo criada pelo controlador principal.

Nesta segunda interação, pode-se observar que o construtor da classe “ProcessoJuridico” precisa verificar se é permitido continuar executando em virtude do controle de acesso.

Esta verificação é necessária para garantir que nenhum objeto de domínio será instanciado, ou terá ser estado alterado por outra classe que não o controlador-fachada principal “GerenciadorProcessoJuridico”.

A classe “ProcessoJuridico” é uma classe do domínio do sistema que trata, ou deveria tratar, somente a responsabilidade de manter o estado de um processo jurídico do mundo real conforme a modelagem orientada a objetos realizada na primeira versão do sistema.

6.1.3 Diagrama de seqüência das telas do sistema

Conforme o requisito “RNF02” (pág. 51), o controle de acesso deverá garantir que apenas usuários autorizados possam acessar as telas do sistema. Um usuário poderá ter privilégios para acessar um determinado conjunto de telas e não ter privilégio para acessar um outro conjunto de telas. Por exemplo, um gerente poderia acessar a tela de consulta de todos os processos jurídicos, mas não poderia acessar a tela de abertura de um processo jurídico porque

este procedimento cabe a responsabilidade do advogado.

Da mesma forma, um advogado poderia consultar todos os advogados registrados no sistema, mas não poderia acessar a tela de inclusão de advogado, procedimento que deve ser executado apenas pelo gerente.

Em um primeiro momento, considerou-se a solução de projetar a verificação do controle de acesso no menu do sistema, o que tornaria dinâmica a exibição ou habilitação dos itens do menu de acordo com os privilégios do usuário autenticado. Contudo, existem telas do sistema original que não são acessadas diretamente pelo menu, por exemplo, a tela de atualização de andamentos do processo jurídico.

A tela de consulta dos andamentos precede o acesso à tela de atualização de andamentos, ou seja, para atualizar um andamento é necessário acessar a tela de consulta dos andamentos, selecionar um andamento da lista de andamentos e clicar no botão “atualizar andamento” que irá encaminhar o usuário à tela de atualização. Suponha-se que o gerente tenha privilégios para consultar os andamentos, mas não para atualizar os andamentos. Neste caso, se faz necessário projetar um controle de acesso em nível de criação de instâncias de tela, e não mais pelos eventos do menu do sistema.

A Figura 25 apresenta o diagrama de seqüência que descreve a interação entre a tela principal do sistema (instância “jFramePrincipal”) e o controlador do controle de acesso na criação e exibição da tela para abertura de processo jurídico.

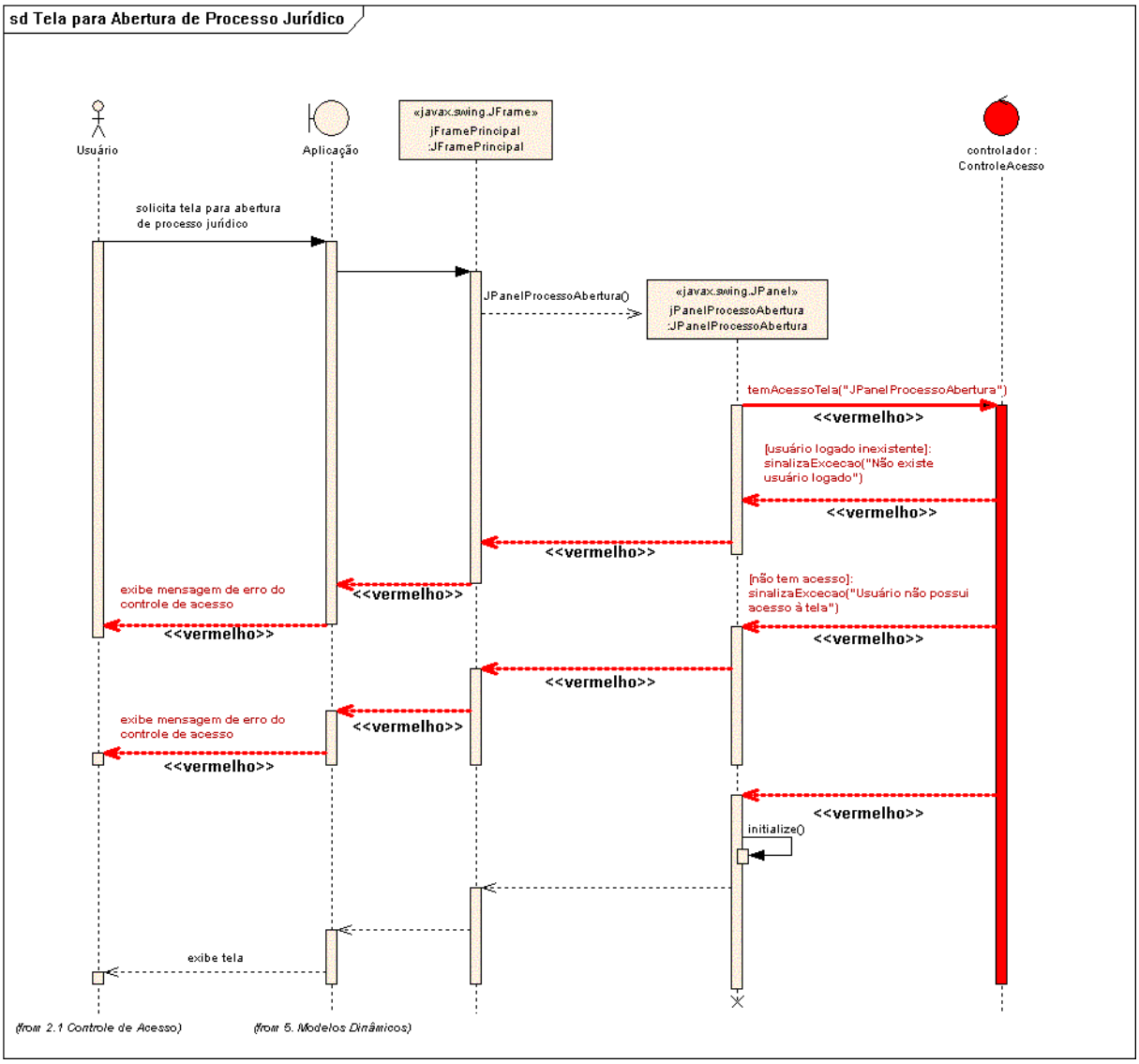


Figura 25 – Diagrama de seqüência na criação da tela para abertura de processo jurídico

6.1.4 Operacionalidade da implementação

Para demonstrar a operacionalidade da implementação do controle de acesso em nível de usuário, a Figura 26 apresenta a tentativa de acesso de um usuário não autorizado à tela de abertura de processo jurídico.

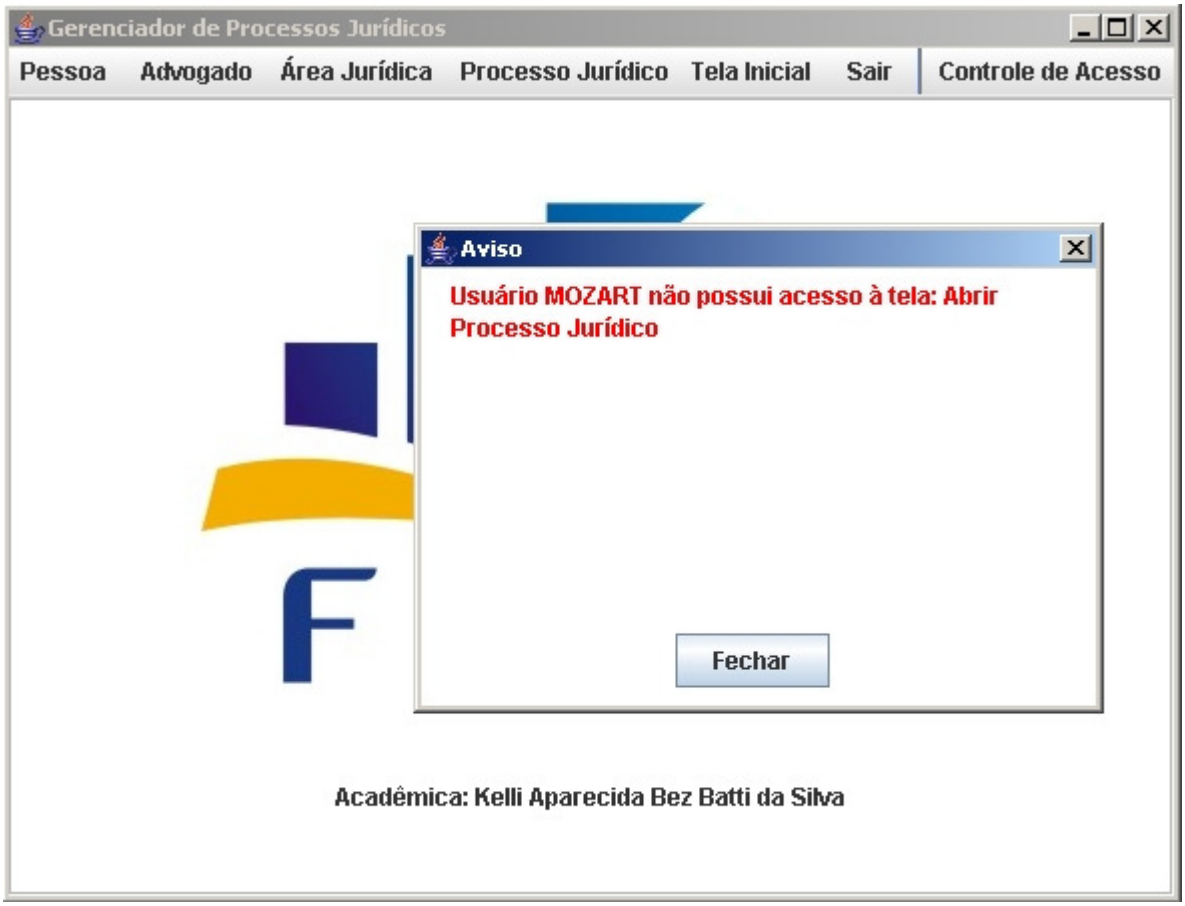


Figura 26 – Erro de controle de acesso ao acessar a tela “Abrir Processo Jurídico”

6.1.5 Considerações finais

Este capítulo demonstrou o impacto de um requisito não funcional em um sistema orientado a objetos. O efeito *crosscuting* gerado pelo requisito ortogonal afetou várias classes do sistema, prejudicando a coesão das mesmas ao atribuir responsabilidades não fortemente relacionadas, aumentando o acoplamento das classes, dificultando a compreensão, manutenibilidade e reusabilidade do sistema.

O cenário apresentado abordou a adição de apenas dois requisitos ortogonais, portanto, um número maior de requisitos desta natureza tornaria as classes cada vez mais complexas e difíceis de manter, compreender e reutilizar.

O próximo capítulo apresenta uma alternativa para o desenvolvimento destes mesmos

requisitos ortogonais de forma a preservar as classes do sistema original e manter o mecanismo de autorização encapsulado em um conjunto de unidades separadas do restante do sistema.

7 DESENVOLVIMENTO DA TERCEIRA VERSÃO – EMPREGO DA POA

Neste ponto do trabalho retoma-se o contexto inicial da segunda versão (pág. 51) onde o cliente solicita o mecanismo de controle de acesso. Contudo, para o desenvolvimento da solução será adotada a Programação Orientada a Aspectos (POA) como alternativa no desenvolvimento deste requisito não funcional.

Para desenvolver esta versão orientada a aspectos, foi necessário modelar os aspectos, definir a responsabilidade de cada aspecto e implementá-los de forma a preservar o sistema original independente da existência destes aspectos.

7.1 ESPECIFICAÇÃO

Durante a modelagem dos aspectos, foram identificados os pontos do sistema (*point cuts*) onde os aspectos deveriam interceptar e aplicar os devidos controles de segurança. As condições impostas (pág. .56) pelos requisitos não funcionais foram atendidas nesta versão, bem como na versão anterior. Três aspectos foram modelados em um diagrama de classes do projeto, conforme é apresentando na Figura 27.

Figura 27 – Diagrama de classes do projeto – Aspectos

O *aspect* “CheckAccessAspect” é responsável pelo controle de acesso às classes de domínio do sistema. Este aspecto não permite que os métodos das classes de domínio sejam executados sem que a origem de sua chamada seja o controlador principal da aplicação e que apenas usuários (ou sistemas) com os devidos privilégios possam acessar as operações deste controlador. O Apêndice A apresenta a implementação do *aspect* “CheckAccessAspect”.

O *aspect* “CheckViewAccessAspect” é responsável pelo controle de acesso às telas do sistema. Este aspecto não permite que as telas do sistema sejam instanciadas por usuários que não possuem privilégios. O Apêndice B apresenta a implementação do *aspect* “CheckViewAccessAspect”.

Já o *aspect* “AuthenticationAspect” controla a existência de um usuário autenticado no sistema. É responsável também pela a exibição da tela de *login* e pelo controle de *logout* no sistema.

Até este ponto foi apresentado o modelo estrutural dos aspectos. A seguir é explanado como estes aspectos alteram o comportamento do sistema.

7.1.1 Diagrama de seqüência dos casos de uso

O comportamento dos eventos do sistema foi alterado sem que fosse necessário alterar as classes do sistema original. A aplicação dos aspectos intercepta pontos definidos no sistema e executa procedimentos restritos apenas ao controle de acesso, como é demonstrado a seguir.

Os diagramas desta seção representam os *join points* interceptados pelos aspectos durante o fluxo de cada cenário do caso de uso "UC04. Manter Processo Jurídico".

A Figura 28 descreve os eventos entre o ator, o sistema e o controlador principal do fluxo "Abertura Processo Jurídico". Em vermelho e azul estão destacados os *point cuts*.

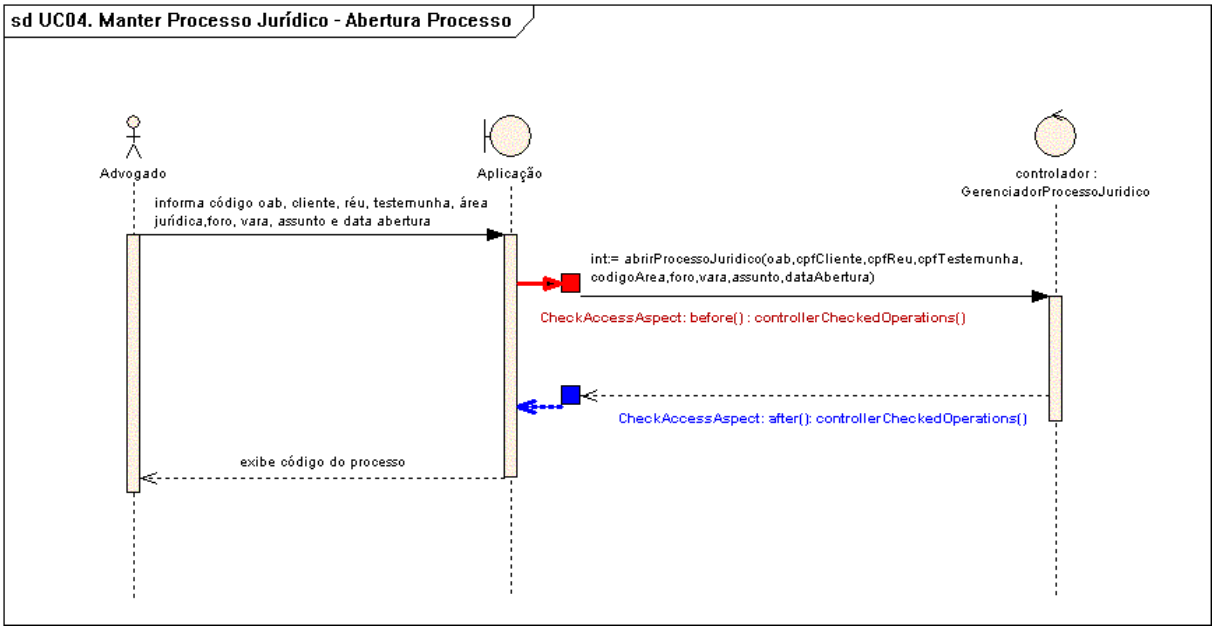


Figura 28 – *Point cuts* no fluxo principal – "Abertura Processo Jurídico".

Neste fluxo, o *aspect* "CheckAccessAspect" intercepta o *join point* (operação de sistema "abrirProcessoJuridico") antes e depois de ser executado o código do método "abrirProcessoJuridico".

7.1.2 Diagrama de seqüência das operações de sistema

O fluxo das operações de sistema foi interceptado pelos *aspects* que, além de garantirem que apenas usuários autorizados executem as operações, monitoram os acessos as classes de domínio.

A Figura 29 descreve a interação entre o sistema, o controlador principal, o controlador do controle de acesso a instância "novoProcessoJuridico" da operação de sistema "abrirProcessoJuridico" . Em vermelho e azul estão destacados os *point cuts*.

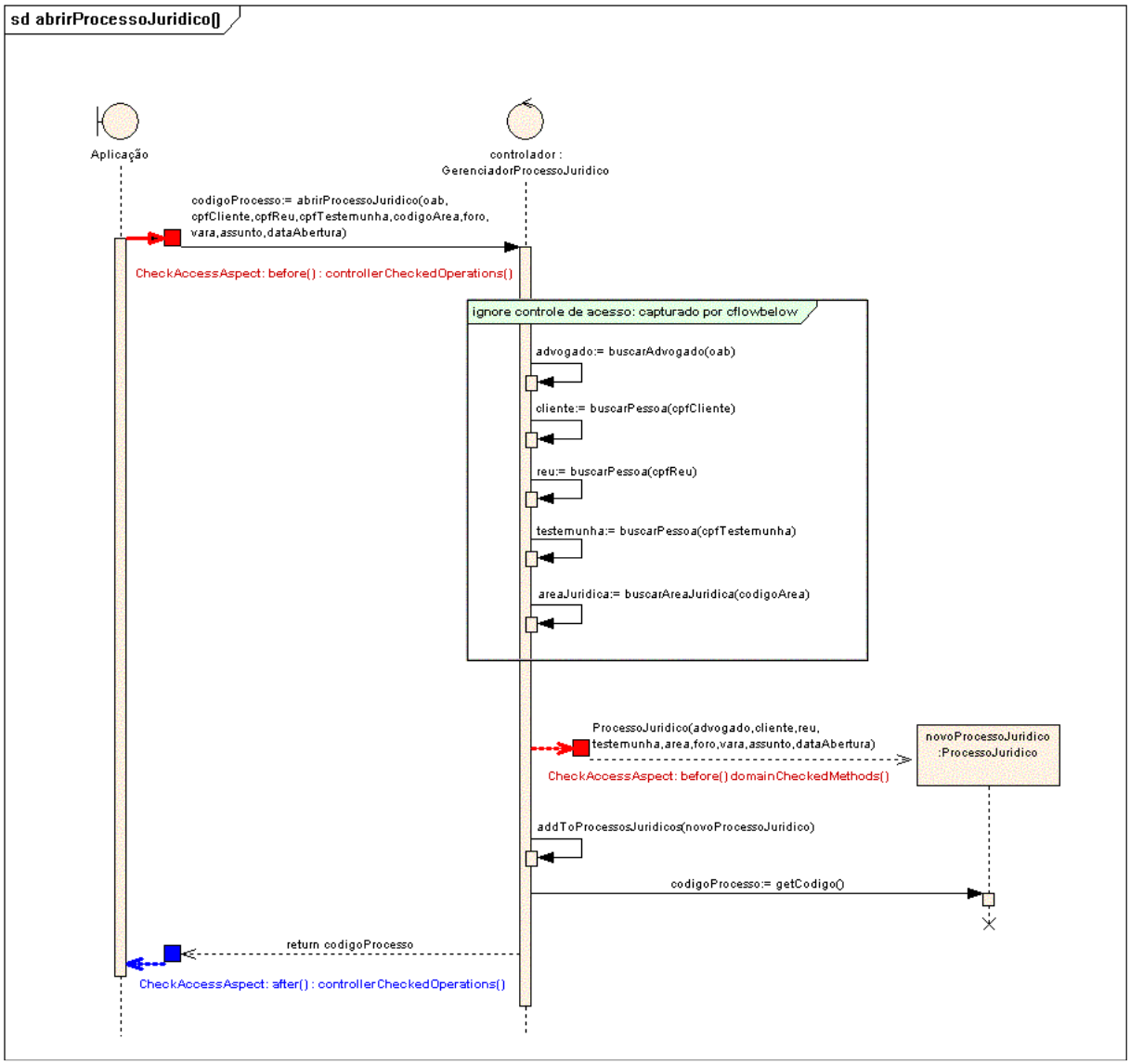


Figura 29 – Pontos interceptados pelos aspectos na operação "abrirProcessoJuridico".

Neste fluxo, o aspecto "CheckAccessAspect" intercepta o fluxo da operação de

sistema “abrirProcessoJuridico” antes e depois de ser executada. Durante o fluxo desta operação, outras operações de sistema são acessadas, porém ignoradas pelo controle de acesso. O objetivo é capturar a primeira operação acessada, se o controle de acesso permitir que esta operação seja executada, todas as operações no qual ela depende serão executadas normalmente. Este mecanismo está identificado pelo fragmento “ignore controle acesso: capturado por *cflowbelow*”. O designador *cflowbelow* do AspectJ captura o fluxo de um *join point*.

A Figura 30 apresenta o código fonte de dois *advices* executados no *join point* capturado pelo *point cut* “controllerCheckedOperations”.

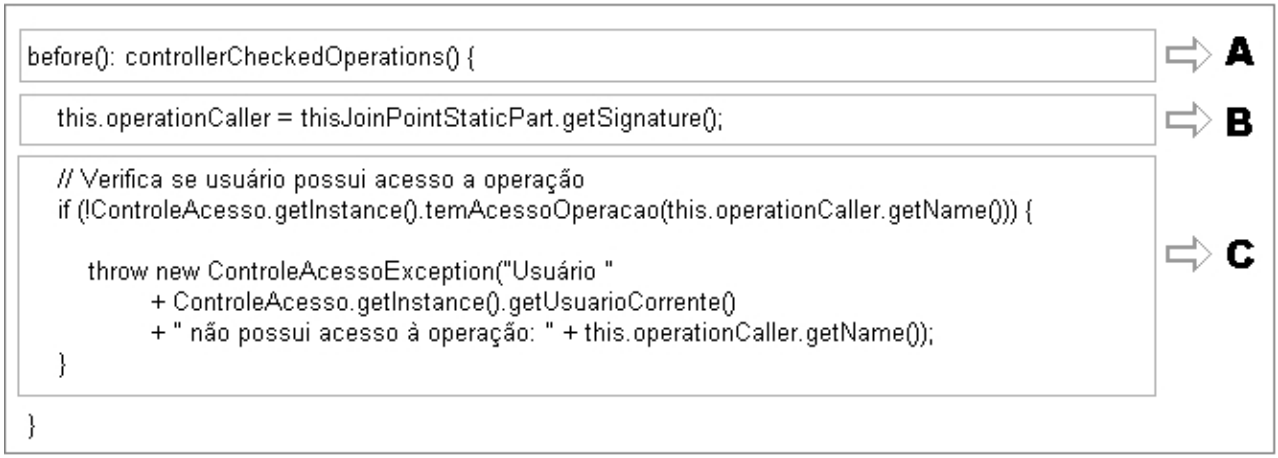


Figura 30 – Código fonte do *advice* executado no *point cut* “controllerCheckedOperations”

O código na região “A” da Figura 30 define que o *advice* deve ser executado antes do *join point* capturado pelo *point cut* “controllerCheckedOperations”. O código na região “B” recupera a assinatura do método do sistema que está sendo interceptado, ou seja, a operação de sistema do controlador que será executada. E o código da região “C” executa a verificação do controle de acesso.

Caso o trecho de código “B” execute com sucesso, o método interceptado executará normalmente como se nada houvesse ocorrido, mas se o usuário não estiver autorizado a acessar a operação, um erro será levantado pelo *advice* e o método interceptado não será executado.

O aspecto “CheckAccessAspect” intercepta também a chamada do construtor da classe “ProcessoJuridico”, através do *pointcut* “domainCheckedMethods()”, para verificar se está sendo chamado do controlador principal. A Figura 31 demonstra o código fonte do *advice* responsável por este processo de verificação.

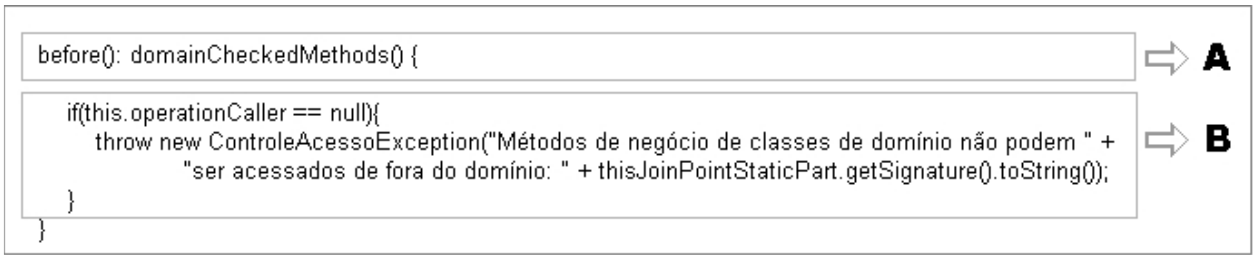


Figura 31 – Código fonte do *advice* executado no *point cut* “domainCheckedMethods”

O código na região “A” da Figura 31 define que o *advice* será executado antes dos *join points* capturados pelo *point cut* “domainCheckedMethods”, e o código na região “B” verifica se o método de domínio está sendo acessado por uma operação do controlador principal “GerenciadorProcessoJuridico”. Caso a condição não seja verdadeira, o *advice* levantará um erro e o método interceptado não será executado, do contrário, o método será executado normalmente.

7.1.3 Diagrama de seqüência das telas do sistema

A Figura 32 apresenta o *point cut* interceptado pelo aspecto "CheckViewAccessAspect" na criação e exibição da tela para abertura de processo jurídico.

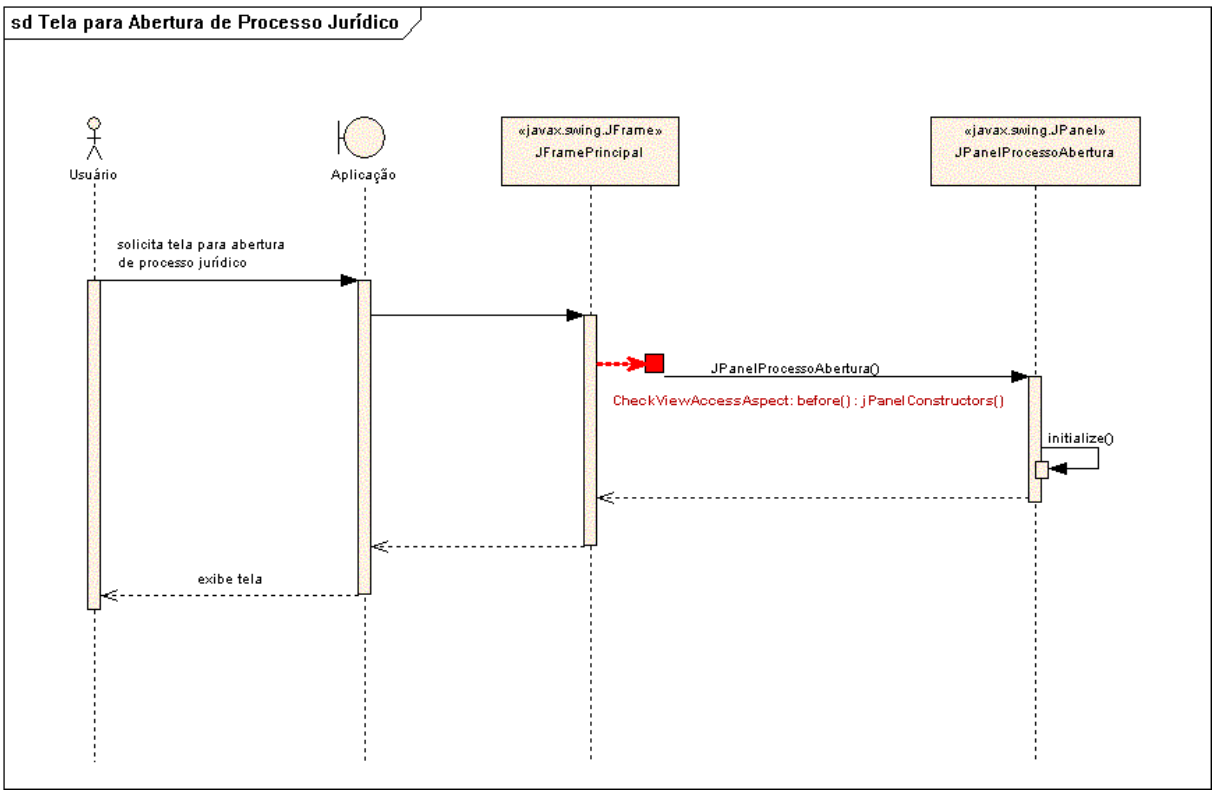


Figura 32 – Point cut na criação da tela para abertura de processo jurídico

7.1.4 Operacionalidade da implementação

Para demonstrar a operacionalidade da implementação do controle de acesso, utilizando programação orientada a aspectos, em nível de usuário, a Figura 33 apresenta a tentativa de acesso de um usuário não autorizado à tela de abertura de processo jurídico.

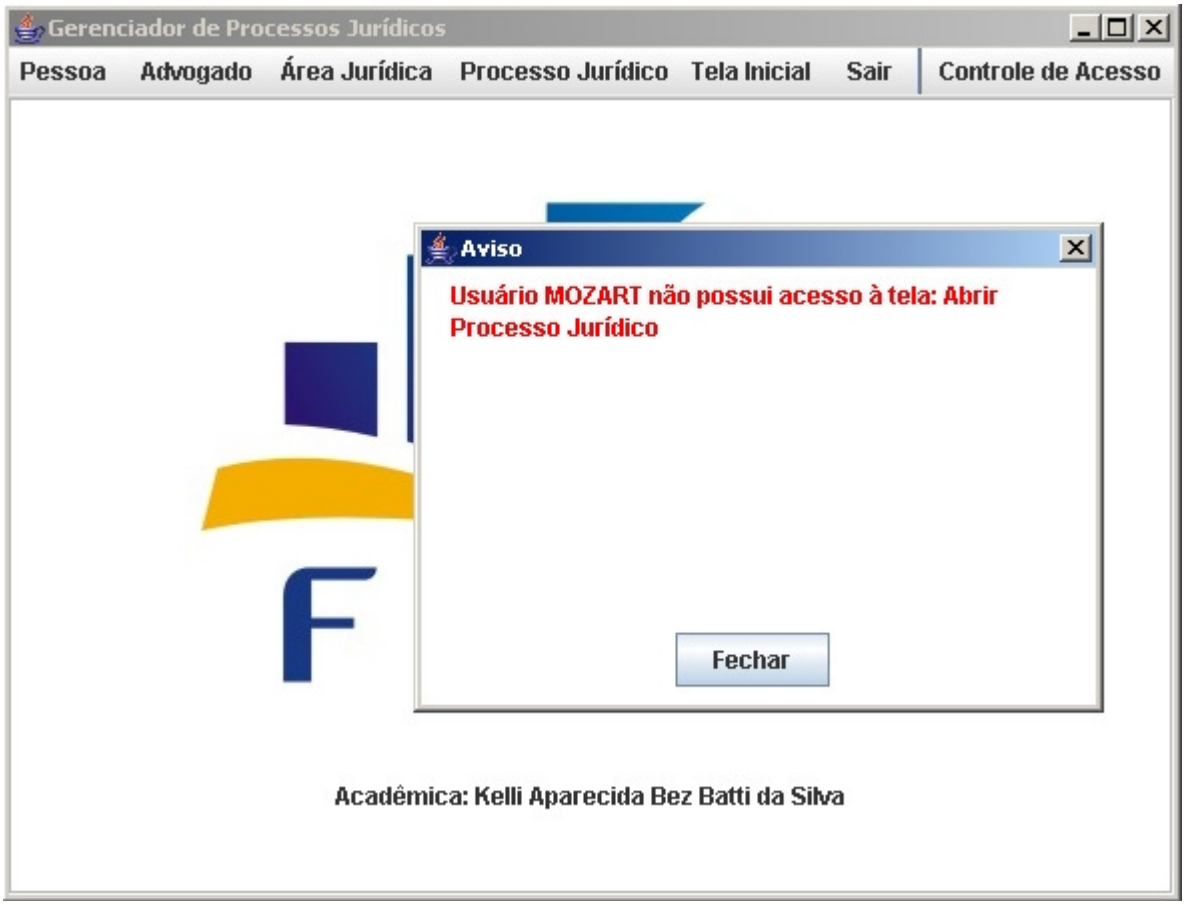


Figura 33 – Erro de controle de acesso ao acessar a tela “Abrir Processo Jurídico”

7.1.5 Considerações finais

Este capítulo apresentou a implementação de um requisito não funcional, em um sistema orientado a objetos, através de programação orientada a aspectos. O efeito *crosscuting* gerado pelo requisito atravessou o comportamento de várias classes do sistema, contudo, sem

a necessidade de alterar o código fonte destas classes e nem prejudicar a coesão das mesmas.

A programação orientada a aspectos, utilizando a linguagem AspectJ, permitiu que se modificasse o comportamento do sistema em determinados pontos através dos *advices* implementados nas classes *aspect.* Esta abordagem encapsulou o mecanismo de controle de acesso em alguns poucos aspectos que satisfizeram as condições impostas pelo requisito.

8 DESENVOLVIMENTO DO TRABALHO – MECANISMO DE TRATAMENTO DE EXCEÇÕES

A linguagem de programação Java especifica, basicamente, dois tipos diferentes de exceções: checadas e não checadas. As exceções checadas obrigam os desenvolvedores a tratarem estas exceções ou a declararem que essas exceções podem ser repassadas adiante. Já as exceções não checadas, não precisam ser trabalhadas de uma forma explícita.

O mecanismo de tratamento de exceções da linguagem AspectJ, chamado *Exception Softening*, permite que exceções checadas sejam convertidas para exceções não checadas e possibilita que estas exceções sejam tratadas dentro de aspectos.

Este recurso funciona basicamente da seguinte maneira: definir quais exceções checadas deverão ser convertidas para não checadas em determinados *point cuts* do sistema. Dessa forma, as exceções serão capturadas e convertidas. A Figura 34 apresenta o código fonte de uma declaração de conversão de exceções.

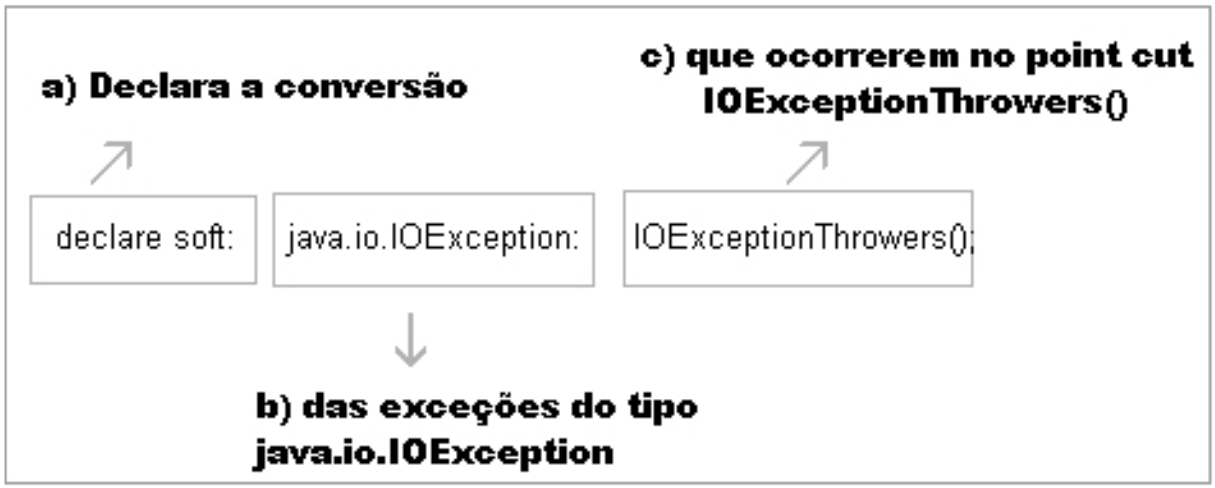


Figura 34 – *Exception Softening* de exceções do tipo "java.io.IOException"

Segundo Laddad (2004, p. 137, tradução nossa), o mecanismo de *exception softening* é um meio prático de evitar que o código de tratamento de exceções se misture à lógica do domínio do sistema, mas deve ser usado cuidadosamente para que as exceções que deveriam estar explicitamente declaradas ou tratadas não sejam mascaradas.

Na terceira versão do sistema desenvolvida neste trabalho, foi adicionalmente aplicado um mecanismo de tratamento de exceções utilizando *exception softening*.

Este mecanismo foi projetado para tratar as exceções que não estão relacionadas ao domínio do sistema, como, por exemplo, as exceções do tipo “java.io.Exception” e “java.lang. ClassNotFoundException”. Estas exceções correspondem a erros na serialização e leitura dos controladores do sistema, em virtude da construção utilizada para persistência de dados do sistema, que consiste em serializar o estado dos controladores.

Optou-se por não converter exceções do tipo “tcc.TCCException”, porque as exceções deste tipo decorrem de erros do domínio do sistema como, por exemplo, não permitir que um andamento do processo jurídico tenha seus dados alterados caso o processo já esteja finalizado, ou que não se permita a existência de mais de uma pessoa com o mesmo cpf.

Estes erros estão explicitamente declarados e tratados nas classes de domínio e no controlador principal do sistema, pois eles fazem parte dos processos de negócio.

A primeira versão do sistema aparentemente não era afetada por requisitos ortogonais, mas o tratamento de determinadas exceções é considerado, pela literatura em geral, aspectos que não deveriam estar misturados ao código da lógica de domínio.

A Figura 35 apresenta o código fonte completo do *aspect* “ExceptionHandlerAspect”, responsável por converter e tratar as exceções do tipo “java.io.Exception”, “java.lang. ClassNotFoundException” e também as exceções do controle de acesso (tcc.ControleAcessoException).

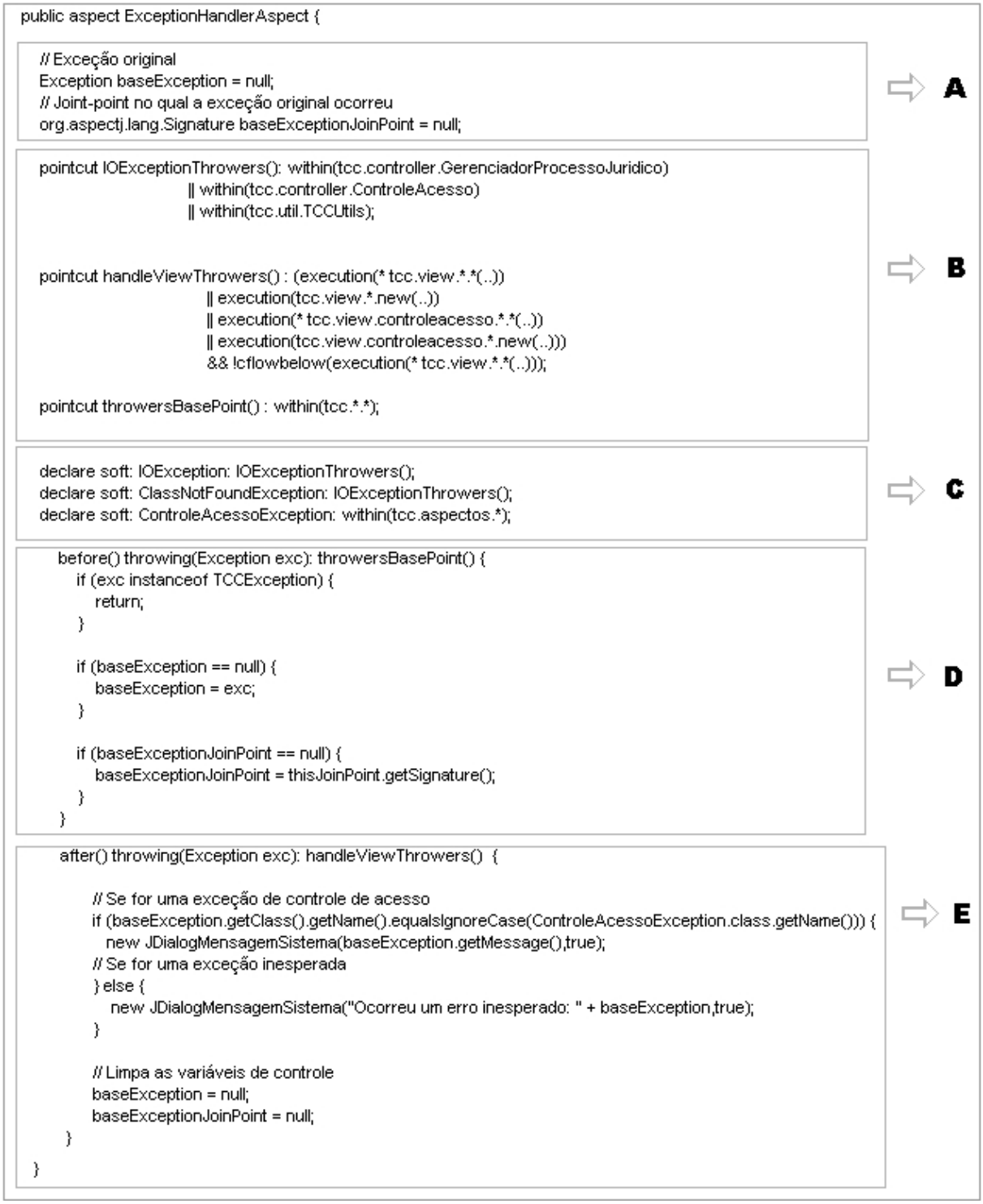


Figura 35 – Implementação do código fonte do *aspect* "ExceptionHandlerAspect"

Na região "A" da Figura 35, são declaradas as variáveis de controle para que se mantenha a exceção original. É possível recuperar a exceção original ao invés da exceção no qual ela foi convertida (org.aspectj.lang.SoftException) através do método "getWrappedThrowable()"da classe "org.aspectj.lang.SoftException". Contudo, um

mecanismo que trata as exceções em um nível mais alto pode perder a referência imediata à exceção original em sua pilha de erros.

Por exemplo, neste trabalho onde as exceções são tratadas no momento em que elas "atingem" a camada de visão (view) do sistema, em boa parte das situações a referência imediata à exceção original é perdida, em função do caminho percorrido pela exceção e pela pilha (*stack trace*) que ela gerou. Frente a isso, definiu-se que a primeira instância de exceção checada que for levantada em qualquer classe do pacote "tcc" (conforme definido no *point cut* "throwersBasePoint" apresentado na Figura 35) será armazenada na variável "baseException". Outra abordagem seria, ao tratar uma exceção, percorrer a pilha até que se encontre uma exceção diferente do tipo "org.aspectj.lang.SoftException", porém, no mecanismo desenvolvido neste trabalho, optou-se por manter a exceção original em uma variável assim que ela ocorrer por uma questão de simplicidade e performance.

A região "B" define os *point cuts* onde as exceções base podem ocorrer e deverão ser armazenadas ("throwersBasePoint"), as classes onde as exceções serão efetivamente tratadas ("handleViewThrowers") e as classes que poderão levantar erros de "IOException" e "ClassNotFoundException" ("IOExceptionThrowers")

Na região "C", são declarados quais tipos de exceções deverão ser convertidos e em quais pontos do sistema. A região "D" contém o código que efetua a armazenagem e controle da exceção original e por fim, na região E, o tratamento das exceções é realizado de acordo com o tipo de exceção ocorrida.

9 ANÁLISE COMPARATIVA ENTRE A SEGUNDA VERSÃO (POO) E A TERCEIRA VERSÃO (POA)

O objetivo principal deste trabalho é a realização de uma análise comparativa entre a Programação Orientada a Objetos (POO) e a Programação Orientada a Aspectos (POA) no desenvolvimento de um requisito não funcional de natureza ortogonal. A análise comparativa avaliou a segunda(pág. 55) e a terceira(pág. 67) versões do sistema no qual um requisito não funcional foi agregado ao sistema original utilizando diferentes abordagens.

Os critérios utilizados na avaliação foram volume de código, funcionalidade da versão final, performance e tempo de desenvolvimento.

A importância de avaliar a diferença de volume de código entre o desenvolvimento nas duas abordagens se deve ao fato de a prática demonstrar que, quanto mais linhas de código forem necessárias para resolver os requisitos de um sistema, maior é a possibilidade de introduzir erros de implementação e, muito provavelmente, tornar o entendimento e a manutenção do sistema proporcionalmente mais complicados.

O código escrito em linguagem Java é interpretado. Embora otimizado através do JIT, o volume de código pode resultar em um número maior ou menor de instruções de baixo nível a serem executadas, conseqüentemente interferindo na performance de execução do sistema.

O aumento de volume de código pode resultar também no aumento de complexidade, em um tempo maior de desenvolvimento, conseqüentemente no aumento de custo e o comprometimento do prazo de entrega de um sistema.

A avaliação de volume de código considerou a métrica MLOC (*Method Lines of Code),* no qual os dados foram apurados por meio do *plugin* "Metrics" que permite avaliar, através da IDE Eclipse, código fonte Java em diversas métricas.

A funcionalidade foi considerada como critério em função de sua importância para o usuário final do sistema. É importante que a abordagem de implementação, seja ela POO ou

POA, preserve a adequação e acurácia do sistema conforme a sua versão original. Neste caso, deve ser transparente para o usuário a adoção de uma nova abordagem de implementação.

Este critério foi avaliado diante de diversos cenários que representam ações que o usuário poderia executar pelo sistema.

Julgou-se importante confrontar a execução da segunda versão com a terceira versão do sistema para identificar possíveis perdas ou ganhos de performance provenientes da programação orientada a aspectos, utilizando a linguagem AspectJ, em relação a programação orientada a objetos, utilizando puramente a linguagem Java. Para realizar esta checagem, utilizou-se o *plugin* "Eclipse Profiler", através da IDE Eclipse, que fornece dados relativos à performance de execução (*CPU Profiling*) e utilização de memória (*Heap*) durante a execução de métodos Java.

Dois pilares determinantes para o resultado de um projeto são custo e prazo. O critério tempo de desenvolvimento foi analisado em razão do impacto que pode causar no custo e no prazo de desenvolvimento de um sistema.

O esforço despendido para produzir um sistema de informação tem custo e, como característica preponderante de um projeto, tem prazo para terminar. Portanto, o custo e o prazo para desenvolvimento de um sistema são proporcionalmente resultantes do tempo empregado na sua produção.

O resultado de cada critério avaliado é apresentado nas seções deste capítulo.

9.1 AVALIAÇÃO DE VOLUME DE CÓDIGO

A avaliação de volume de código demonstrou resultados significativamente favoráveis à programação orientada a aspectos na implementação de requisitos ortogonais.

Para a implementação do requisito de autorização nas classes de domínio e no

controlador, houve uma redução de aproximadamente 73% no volume de código encapsulado no aspecto “CheckAccessAspect” em relação ao código implementado em POO que ficou espalhado pelas classes do sistema.

Esta expressiva redução de código fonte se deve ao fato de que, aplicando a programação orientada a objetos, todas as operações de sistema do controlador continham uma chamada ao controle de acesso para verificar se a execução era autorizada e, não obstante ao controlador, as classes de domínio também continham chamadas para o controle de acesso dentro de seus construtores e métodos de negócio.

Com o emprego da programação orientada a aspectos, foi desenvolvido um *aspect* que intercepta o controlador e as classes de domínio para garantir o controle de acesso sem incluir, de fato, qualquer linha de código fonte nestas classes referente ao controle de acesso.

A Figura 36 apresenta um gráfico que revela a representatividade, em volume de código, do requisito de controle de acesso em relação ao código da lógica de domínio e do controlador da primeira versão do sistema.

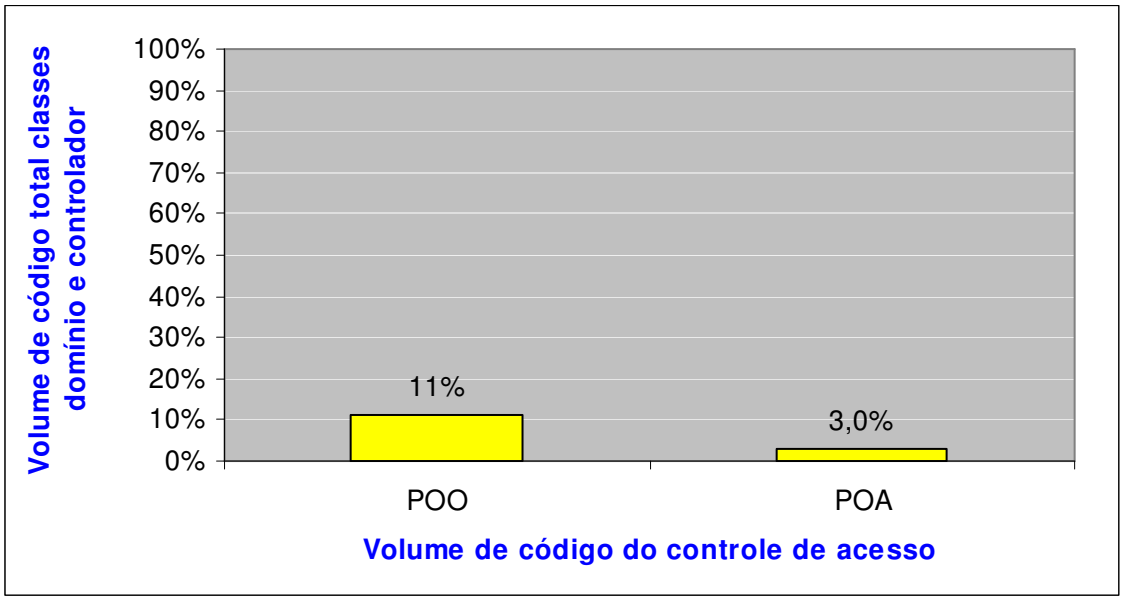


Figura 36 – Representatividade do volume de código do controle de acesso em relação ao sistema original

Este gráfico demonstra que 11% do código contido nas classes de domínio e no controlador do sistema é decorrente do controle de acesso implementado com POO. Ao

confrontar o volume de código do *aspect* “CheckAccessAspect” em relação ao volume de código das classes de domínio e do controlador (que são afetadas por este *aspect*), constatou-se que o código encapsulado pelo aspecto representa apenas 3%. Com estes dados é possível concluir que, além de evitar o espalhamento de código por entre as classes, a implementação do *aspect* resultou em um volume de código significativamente menor do que a programação orientada a objetos.

A Figura 37 apresenta um gráfico que demonstra, através da métrica MLOC (*Method Lines of Code*), a diferença de volume de código dentro dos métodos do controlador principal “GerenciadorProcessoJuridico” após a implementação do controle de acesso às operações de sistema, ou seja, dos métodos do controlador.

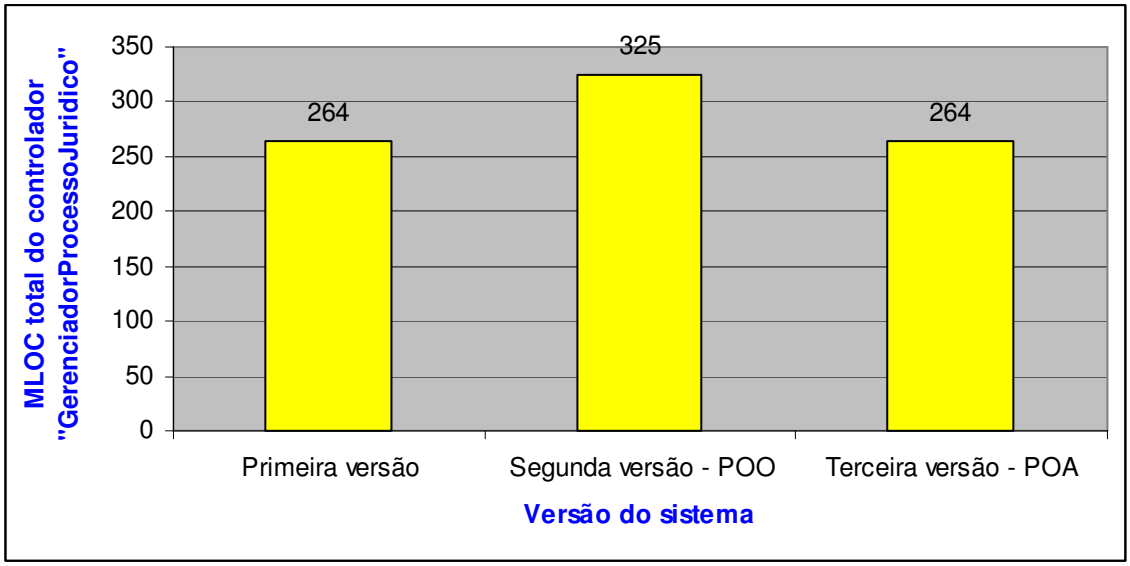


Figura 37 – Diferença de MLOC no controlador “GerenciadorProcessoJuridico”

Observa-se no gráfico que a terceira versão do sistema preservou o controlador “GerenciadorProcessoJuridico” sem qualquer alteração em relação a primeira versão. Já a segunda versão, resultou em um acréscimo de linhas de código dentro dos métodos do controlador. O controlador desenvolvido na primeira versão e na terceira versão possui, cada um, 264 linhas de código fonte. Já o controlador desenvolvido na segunda versão possui 325 linhas de código fonte, resultando em 61 linhas de código incluídas para garantir o controle de acesso às operações de sistema.

A implementação da autorização nas telas do sistema em POA apresentou uma redução de 36% no volume de código encapsulado no aspecto “CheckViewAccessAspect” em relação ao código implementado em POO que ficou espalhado pelas telas do sistema.

O impacto da redução de volume código no tratamento de exceções alcançado com a terceira versão do sistema, em uma comparação com a segunda versão, foi de aproximadamente 92% em relação à declaração de exceções na assinatura dos métodos (*throws*) e 85% nos blocos de tratamento (*try/catch*) que ficaram espalhados pelas classes do sistema.

Em relação à versão original do sistema, a segunda versão afetou a grande maioria das classes do sistema de forma invasiva, misturando código da escrita original das classes com o código necessário para satisfazer os novos requisitos funcionais.

Em contrapartida, a terceira versão do sistema, se comparada à versão original, preservou as classes de domínio, o controlador-fachada e as telas sem qualquer alteração.

Para isso, foram implementados quatro aspectos que afetaram o comportamento do sistema, em pontos definidos nos próprios aspectos, para as verificações do controle de acesso e no tratamento de exceções não relacionadas aos processos de negócio.

9.2 AVALIAÇÃO DE FUNCIONALIDADE DA VERSÃO FINAL

A avaliação da funcionalidade final teve como princípio considerar a visão do usuário em relação ao sistema. Com este objetivo, foram realizados testes em diferentes cenários para verificar se as versões subseqüentes do sistema preservaram a acurácia e a adequação das rotinas já existentes.

Entende-se por acurácia, “atributos do software que evidenciam a geração de resultados ou efeitos corretos ou conforme acordados” (ROCHA, 2002). Entende-se por

adequação, “atributos de software que evidenciam a presença de um conjunto de funções e sua apropriação para as tarefas especificadas.”. (ROCHA, 2002)

Observou-se que ambas versões operam de forma idêntica nos cenários avaliados, conforme descreve o Quadro 6:

Cenários	POO	POA
Acesso autorizado às telas do sistema.	Usuário encaminhado com sucesso para a tela.	Usuário encaminhado com sucesso para a tela.
Acesso não autorizado às telas do sistema.	Abortou a navegação e exibiu mensagem de erro ao usuário.	Abortou a navegação e exibiu mensagem de erro ao usuário.
Acesso autorizado às operações do sistema na execução direta via método *main()* de uma classe de teste.	Operação foi executada.	Operação foi executada.
Acesso não autorizado às operações do sistema na execução direta via método *main()* de uma classe de teste.	Operação foi abortada e um erro foi levantado pela aplicação.	Operação foi abortada e um erro foi levantado pela aplicação.
Acesso direto aos métodos de negócio de domínio, sem participação do controlador-fachada .	Operação foi abortada e um erro foi levantado pela aplicação.	Operação foi abortada e um erro foi levantado pela aplicação.
Realização da abertura de um Processo Jurídico. (Acesso autorizado)	Operação realizada conforme especificação original (1ª versão)	Operação realizada conforme especificação original (1ª versão)
Realizar demais cadastros no sistema. (Acesso autorizado)	Operação realizada conforme especificação original (1ª versão)	Operação realizada conforme especificação original (1ª versão)

Quadro 6 – Avaliação da funcionalidade final

9.3 AVALIAÇÃO DE PERFORMANCE

Para realizar a avaliação de performance foi utilizado o *plugin* “Eclipse Profiler”, que permite monitorar a performance de métodos Java durante a execução.

A operação de sistema “abrirProcessoJuridico” foi executada 10.000 (dez mil) vezes para se obter o tempo de execução da operação em uma escala significativa. Observou-se que a terceira versão do sistema apresenta uma pequena defasagem de performance em relação à segunda versão.

Na Figura 38 é apresentado um gráfico que demonstra, na grandeza de milisegundos, o tempo gasto na execução da operação em ambas as versões do sistema, POO e POA.

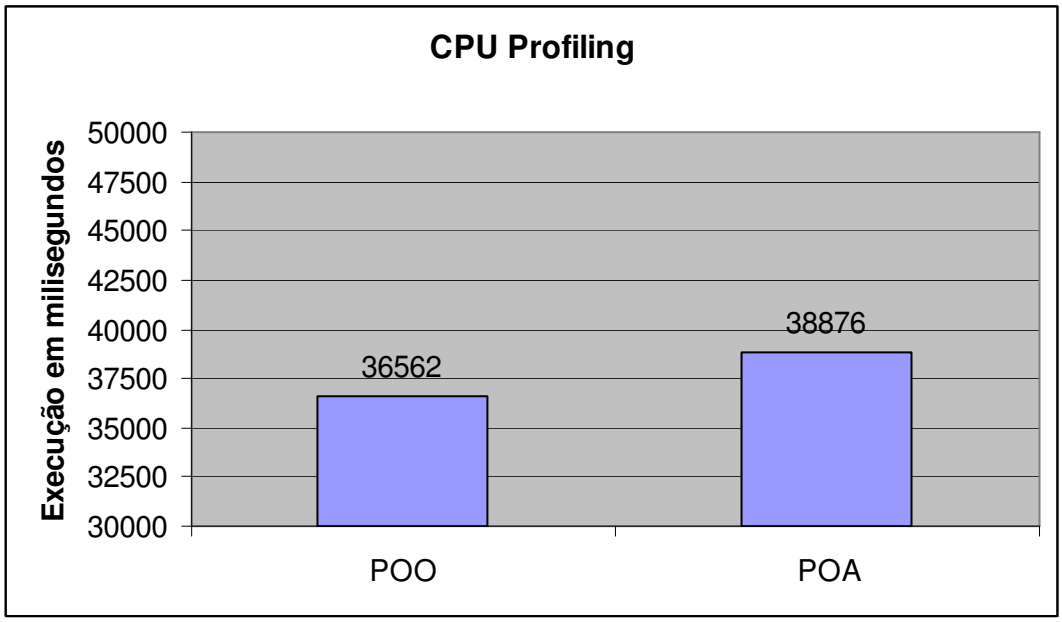


Figura 38 – CPU Profiling

O resultado desta análise evidencia uma diferença de 2,3 segundos a mais na execução da operação “abrirProcessoJuridico” da terceira versão (POA), em relação a segunda versão (POO).

Contudo, uma melhor ou pior performance está diretamente relacionada à escrita do código dos *advices*, segundo estudos de membros da comunidade AspectJ, relatados em Hilsdale (2004).

O resultado das pesquisas apresentadas em Hilsdale (2004), comprovou também que, ao modificar determinado código do AspectJ utilizado em implementações de requisitos de *loggin,* foi possível reduzir em 76% o tempo de execução em relação a mesma aplicação desenvolvida apenas com uma abordagem menos flexível com código Java distribuído por entre as classes.

9.4 AVALIAÇÃO DE TEMPO DE DESENVOLVIMENTO

Com a experiência obtida no desenvolvimento deste trabalho, há uma forte expectativa de ganho de produtividade na implementação de requisitos ortogonais através da programação orientada a aspectos. Em função do estudo da linguagem AspectJ e do esclarecimento de alguns conceitos durante a aplicação desta abordagem a este trabalho, uma avaliação de esforço despendido poderia ser equivocada, atribuindo o tempo gasto em investigações iniciais sobre a escrita de código com esta linguagem e frente as dificuldades encontradas no decorrer do desenvolvimento face a pouca experiência na mesma, distorcendo o resultado da avaliação deste critério ao comparar um conhecimento mais solidificado (POO) em relação a um conhecimento explorado durante o desenvolvimento do trabalho.

Atualmente, com uma maior experiência de desenvolvimento nesta linguagem, é possível afirmar que o ganho de produtividade é um forte potencial da programação orientada a aspectos que, como qualquer outra tecnologia, exige uma curva de aprendizado. Contudo, este desvio não foi, de forma alguma, desmotivante em relação a POA, pois foram localizadas diversas referências de fácil compreensão e adequadas à aprendizagem e ao suporte à linguagem.

9.5 RESULTADOS E DISCUSSÃO

Como se pode observar, o trabalho baseou-se em um estudo de caso fictício tendo em vista que o foco principal foi o estudo comparativo entre POO e POA. Além disso, com o advento dos novos requisitos não funcionais que foram aplicados à versão original do sistema, houve a necessidade de desenvolver parte de um módulo de controle de acesso, incorporado ao sistema, para possibilitar a administração de usuários e privilégios a servirem como base para o controle de acesso do sistema.

A etapa de análise comparativa avaliou algumas características importantes a serem consideradas na adoção de uma nova metodologia e linguagem de programação. Foi possível, através dos testes realizados, avaliar a funcionalidade final e o volume de código de cada versão e obter os resultados apresentados no capítulo 8 (pág. 80).

O Quadro 7 relaciona os trabalhos correlatos mencionados na fundamentação teórica (pág. 35) com este trabalho, identificando características em comum e distintas em cada um dos trabalhos.

Características	**Este trabalho**	**Soares, Borba (2002)**	**Gradecki, Lesiecki (2003),**	**Hugo, Grott (2005)**
Análise, projeto e implementação orientados a objetos.	X	X		X
Programação orientada a aspectos com AspectJ	X	X	X	X
Mecanismo de autorização	X		X	
Mecanismo de tratamento de exceções	X		X	
Mecanismo de *loggin*			X	X
Análise comparativa entre POO e POA	X	X	X	

Quadro 7 – Características em comum e distintas de cada trabalho

10 CONCLUSÕES

Os resultados obtidos com este trabalho demonstram a necessidade e a importância da exploração de alternativas, como a programação orientada a aspectos, para o progresso do desenvolvimento de software frente a amplitude e complexidade das necessidade a serem atendidas pelos sistemas atuais.

A existência de requisitos ortogonais de software é muito comum e, geralmente, resultam em soluções deselegantes e onerosas, dificultando o trabalho dos profissionais que precisam se preocupar, desde os momentos iniciais de um projeto, com propriedades desvinculadas do propósito principal do sistema.

Foi possível observar, na prática, algumas limitações da orientação a objetos na implementação de um requisito ortogonal de controle de acesso, no qual a implementação orientada a objetos ocasionou a perda de coesão e o aumento do acoplamento das classes afetadas por este requisito, além de resultar em um aumento expressivo no volume de código fonte.

Os objetivos deste trabalho foram alcançados e a realização da análise comparativa trouxe, de forma satisfatória, a percepção das vantagens do emprego da orientação a aspectos no suprimento de deficiências da orientação a objetos quanto à implementação de requisitos ortogonais.

10.1 EXTENSÕES

Como sugestão para trabalhos futuros na área de programação orientada a aspectos, recomenda-se:

a) realizar uma análise comparativa entre programação orientada a aspectos e

reflexão computacional.

b) implementar cenários de um estudo de caso aplicando o padrão *wormhole* (LADDAD, 2003, p. 256)

c) sistematizar a análise comparativa direcionando-a ao formato de engenharia de software experimental, considerando variáveis, hipóteses, participantes, contexto do experimento, entre outros.

d) incluir novos critérios de comparação.

REFERÊNCIAS BIBLIOGRÁFICAS

CHAVEZ, Christina von Flach Garcia; LUCENA, Carlos José Pereira de.**Um enfoque baseado em modelos para o design orientado a aspectos**. 2004. 298 f. Tese de Doutorado (Pós-Graduação em Informática) - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

CLEMENT, Andy; KERSTEN, Mik. **Aspect-oriented programming with aspectJ**. [Burlingame], 2005. Disponível em <http://www.eclipsecon.org/2005/tutorials.php>. Acesso em: 26 out. 2004.

DEITEL, H. M.; DEITEL, P.J.**Java**: como programar. Tradução Carlos Arthur Lang Lisboa. 4.ed. Porto Alegre: Bookman, 2003.

DEXTRA SISTEMAS. **Entendendo a programação orientada a aspectos**. São Paulo, [2005]. Disponível em <http://www.dextra.com.br/empresa/artigos/aspectprog.htm>. Acesso em: 24 out. 2005.

GAMMA, Erich, et al. **Design patterns**: elements of reusable object-oriented software. Reading,Massachusetts: Addison-Wesley, 1994. 395p.

GRADECKI, Joseph D.; LESIECKI, Nicholas. **Mastering aspectJ**: aspect-oriented programming in java. Indiana:Wiley Publishing, 2003.

GROTT, Marcio Carlos; MARCEL, Hugo. **Estudo de caso aplicando programação orientada a aspecto**. In: SEMINCO, 14., 2005, Blumenau. Anais... Blumenau: Furb, 2005. p. 45-56.

HILSDALE, Erik ; HUGUNIN, Jim. **Advice weaving in aspectJ**. In: AOSD, 2004, Lancaster. Anais... Lancaster: [s.n.], 2004. p. 1-10.

KICZALES, G. et al. **An overview of aspectJ**. Proceedings of the European Conference on Object-Oriented Programming (ECOOP'01), Budapest, Hungary, 2001.

LADDAD, Rammivas. **AspectJ in action**. Greenwich: Manning Publications Co, 2003.

LARMAN, Craig. **Aplicando UML e padrões**: uma introdução à análise e projeto orientados a objetos. Tradução Luiz Augusto Meirelles Salgado e João Tortello. 2.ed. Porto Alegre: Bookman, 2004.

LEITE, Mario; RAHAL JUNIOR, Nelson Abu Sanra. **Programação orientada ao objeto**: uma abordagem didática. São Paulo, 2002. Disponível em <http://www.revista.unicamp.br/infotec/>. Acesso em: 31 março 2005.

O'REGAN, Graham. **Overview of aspect oriented programming**, 2004. Disponível em <http://www.onjava.com/pub/a/onjava/2004/01/14/aop.html>. Acesso em: 30 março 2005.

ROCHA, Ana Regina**. Qualidade de software**: processo e produto. In: ENCONTRO DA QUALIDADE E PRODUTIVIDADE EM SOFTWARE, 2002, Petrópolis. Disponível em <http://www.mct.gov.br/Temas/info/Dsi/PBQP/ReuniaoPetropolis/Palestra COPPE.pdf>. Acesso em: 24 jan. 2006.

SANTOS, Márcia; LENCASTRE, Maria; CASTRO, Jaelson Brelaz; FONSECA, Décio. **O uso do framework NFR no projeto de banco de dados**. In: WORKSHOP DE ENGENHARIA DE REQUISITOS, 3., 2000, Rio de Janeiro. Anais... : [S.l.: s.n.], 2000. p. 210.

SINTES, Anthony. **Aprenda programação orientada a objetos em 21 dias**. Tradução João Eduardo Nóbrega Tortello. São Paulo: Pearson Education do Brasil, 2002.

SOARES, Sérgio; BORBA, Paulo. **AspectJ - programação orientada a aspectos em java**. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 6., 2002, Rio de Janeiro. Anais... Rio de Janeiro: [s.n.], 2002. p. 1-17.

TIRELO, Fabio , et al. **Desenvolvimento de software orientado por aspectos**. In: JAI - JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, 23., 2004, Salvador. Anais... Belo Horizonte: [s.n.], 2004. p. 3-39.

WAZLAWICK, Raul Sidnei. **Análise e projeto de sistemas de informação orientado a objetos**. Rio de Janeiro: Elsevier, 2004. 298p.

APÊNDICE A – Implementação do aspect “CheckAccessAspect”

```
public aspect CheckAccessAspect {


    protected Signature operationCaller;

    pointcut domainConstructors() : execution(tcc.model.*.new(..));

    pointcut domainGetMethods() : execution(public * tcc.model.*.get*(..))
         || execution(public * tcc.model.*.is*(..))
         || execution(public * tcc.model.*.toString(..))
         || execution(public * tcc.model.controleacesso.*.temAcesso*(..));

    pointcut domainSetMethods() : execution(public * tcc.model.*.set*(..))
                        || execution(public * tcc.model.*.addTo*(..))
                        || execution(public * tcc.model.*.removeFrom*(..));

    pointcut domainCheckedMethods() : (execution(* tcc.model.*.*(..))
                                    || domainConstructors())
                                    && !domainGetMethods()
                                    && !domainSetMethods();
    pointcut controllerCheckedOperations():
             execution(* tcc.controller.*.*(..))
             && !cflowbelow(execution(* tcc.controller.*.*(..)))
             && !execution(static * tcc.controller.*.*(..))
             && !execution(* tcc.controller.*.serializar(..))
             && !execution(* tcc.controller.ControleAcesso.temAcesso*(..))
             && !execution(* tcc.controller.ControleAcesso.cargaInicial())
             && !execution(* tcc.controller.ControleAcesso.buscar*(..))
             && !execution(private * tcc.controller.*.*(..))
             && !execution(public * tcc.controller.*.get*(..))
             && !execution(public * tcc.controller.*.set*(..))
             && !execution(public * tcc.controller.*.login(..))
             && !execution(public * tcc.controller.*.logout());

    before(): controllerCheckedOperations() {
        this.operationCaller = thisJoinPointStaticPart.getSignature();
        // Verifica se usuário possui acesso a operação
        if (!ControleAcesso.getInstance().temAcessoOperacao(
                this.operationCaller.getName())) {
            throw new ControleAcessoException("Usuário "
                    + ControleAcesso.getInstance().getUsuarioCorrente()
                    + " não possui acesso à operação: "
                    + this.operationCaller.getName());
        }
    }

    after(): controllerCheckedOperations() {
      this.operationCaller = null;
    }

    before(): domainCheckedMethods() {
      if(this.operationCaller == null){
          throw new ControleAcessoException("Métodos de negócio de classes
            de domínio não podem ser acessados de fora do domínio: " +
            thisJoinPointStaticPart.getSignature().toString()); } }
}
```

APÊNDICE B – Implementação do aspect “CheckViewAccessAspect”

```
public aspect CheckViewAccessAspect {

    pointcut jPanelConstructors() : (execution(tcc.view.*.new(..))
                    || execution(tcc.view.controleacesso.*.new(..)))
                    && !execution(tcc.view.JFramePrincipal.new(..))
                    && !execution(tcc.view.JDialogMensagemSistema.new(..))
                && !execution(tcc.view.controleacesso.JPanelLogin.new(..));


    pointcut jFrameSetContent(Container arg0) :
            call(* tcc.view.JFramePrincipal.setContentPane(Container))
              && args(arg0);

    before(): jPanelConstructors() {

        // Verifica se usuário possui acesso a operação
        String jPanelName =
thisJoinPointStaticPart.getSignature().getDeclaringType().getSimpleName();
        ControleAcesso controlador = ControleAcesso.getInstance();

        if (!controlador.temAcessoTela(jPanelName)) {
            Tela tela = controlador.buscarTela(jPanelName);
            throw new ControleAcessoException("Usuário " +
            ControleAcesso.getInstance().getUsuarioCorrente()  + " não
            possui acesso à tela: " + tela.getDescricao());
        }
    }

    void around(Container arg0): jFrameSetContent(arg0) {
        ControleAcesso controlador = ControleAcesso.getInstance();

        String tela = arg0.getClass().getSimpleName();
        if (tela.equals("JPanelLogin") || tela.equals("JPanel") ||
            controlador.buscarTela(tela) ==    null ||
            controlador.temAcessoTela(tela)) {
            proceed(arg0);
        }
    }
}
```