UNIVERSIDADE FEDERAL DE PERNAMBUCO
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
CENTRO DE INFORMÁTICA

REFATORAÇÃO DE SISTEMAS ORIENTADOS A OBJETOS PARA AUXILIAR A CONSTRUÇÃO DE UM TESTBED PARA MANUTENÇÃO DE SOFTWARE ORIENTADO A ASPECTOS

TRABALHO DE GRADUAÇÃO

Aluno: Antônio Carlos da Silva Júnior (acsj@cin.ufpe.br)
Orientador: Sérgio Castelo Branco Soares (scbs@cin.ufpe.br)

Recife, Dezembro de 2011

Resumo

Existem, atualmente, muitas tecnologias em desenvolvimento, porém algumas delas não são utilizadas em projetos reais. Isto acontece, principalmente, porque tais tecnologias foram pouco experimentadas e testadas, causando assim, um obstáculo para sua utilização na indústria de software. Assim, uma abordagem bastante utilizada por pesquisadores é a realização de experimentos para avaliar o impacto do uso e a significância das *features* de uma dada tecnologia. Para tentar reduzir os riscos desses experimentos e dar mais credibilidade aos estudos realizados, a engenharia de software utiliza-se de testbeds.

Testbeds são compostos por um conjunto de artefatos associados a um sistema, além da infraestrutura necessária para a realização dos estudos. O objetivo deste trabalho é demonstrar os benefícios associados ao uso da programação orientada a aspectos (POA) por meio de um estudo empírico. Para isso, serão realizadas uma série de refatorações utilizando POA num sistema desenvolvido puramente em programação orientada a objetos (POO) e, posteriormente, seus resultados serão avaliados. Além disso, para garantir que o comportamento do sistema original não sofra alterações, serão realizados casos de teste no contexto de cada área modificada.

Assim, será desenvolvida uma estrutura para servir como suporte para execução e replicação de estudos posteriores. Deste modo, ambos os sistemas servirão como elementos de testbed para pesquisadores utilizarem-nos em seus experimentos e estudos empíricos.

Agradecimentos

Agradeço, primeiramente, a Deus por ter me dado saúde, força e capacidade para concluir este trabalho, pois sem Ele nada seria possível. À Deus toda honra e toda glória.

Um agradecimento especial aos meus pais Antônio e Rosa e às minhas irmãs Nel e Carla pelo amor, amizade e paciência ininterruptos. Agradeço por sempre estarem ao meu lado incentivando meus estudos.

Gostaria de agradecer, também, à minha namorada Clislainy pelo apoio que sempre tem me dado ao longo de todo este percurso.

À Juliana Saraiva pela paciência, conversas e dicas que muito me ajudaram, inclusive revisando este trabalho e sugerindo melhorias.

Ao meu orientador Sérgio Soares pela oportunidade e credibilidade para a elaboração deste trabalho.

Aos professores e funcionários do Centro de Informática (CIn), pelos ensinamentos e apoio.

Aos meus amigos do g10 pelas conversas, brincadeiras, ajudas e companhia nas viradas de noite durante toda essa jornada.

Enfim, à todos que de alguma forma contribuíram para a conclusão deste trabalho.

Sumário

1. Introdução

Uma das grandes dificuldades encontradas por pesquisadores é o fato de muitas metodologias e ferramentas desenvolvidas na academia não serem adotadas pela indústria de Tecnologia de Informação e Comunicação (TIC). Apesar de serem notórios os benefícios trazidos por grande parte dessas novas tecnologias, muitas empresas e instituições de desenvolvimento de software ainda se sentem inseguras para aderirem a novos paradigmas.

De acordo com Redwine e Riddle [16], uma tecnologia emergente pode levar de 15 a 20 anos para amadurecer e ser amplamente utilizada. Assim como outras novas tecnologias, o Desenvolvimento Orientado a Aspectos (DSOA, do inglês, ou AOSD - Aspect Oriented Software Development) também tem encontrado dificuldades em se difundir pelo mercado. Segundo pesquisa realizada em [9], apesar da Programação Orientada a Aspectos (POA, do inglês, AOP - Aspect Oriented Programming) ser de conhecimento da maioria das empresas, ela ainda não é uma realidade como padrão de desenvolvimento por não terem sido identificados ganhos ou perdas que justificassem sua adoção.

A POA tem sido apresentada como uma tecnologia muito útil para a engenharia de software em vários cenários, pois provê uma melhor forma de lidar com problemas de domínio [1]. Porém, ainda há uma barreira a ser ultrapassada para que seu uso seja realmente difundido pela indústria de software. Um dos principais motivos para a existência dessa barreira é a falta de estudos empíricos sistemáticos que esclareçam seus riscos e comprovem os benefícios associados.

1.1 Motivação

Apesar de muitos frameworks propostos pela comunidade ajudarem os pesquisadores em seus estudos, algumas tecnologias ainda precisam de guias muito específicos para serem avaliadas [34]. Estes guias fornecem informações específicas para que seja tirado o máximo proveito da tecnologia em questão.

Para isso, se faz necessária a existência de uma infraestrutura que facilite a condução desses experimentos. Com o intuito de reduzir os riscos desses experimentos e dar mais credibilidade aos estudos realizados, a engenharia de software utiliza-se de testbeds.

Testbeds são ambientes empíricos nos quais pesquisadores podem planejar e executar estudos com a intenção de avaliar e/ou comparar tecnologias [34]. Testbeds são compostos por um conjunto de artefatos associados a um sistema, além da infraestrutura necessária para a realização dos estudos [15]. Então, a partir dos testbeds é possível analisar e avaliar os resultados das experimentações para a comprovação dos benefícios e aplicabilidade de uma dada tecnologia como DSOA.

1.2 Objetivos

Neste trabalho, foi utilizado como objeto de estudo o projeto Amadeus[1] [3] que utiliza unicamente programação orientada a objetos (POO). Nele, foi realizada uma série de refatorações com o intuito de agregar recursos da programação orientada a aspectos (POA) gerando, assim, um sistema com uma nova arquitetura, porém com funcionalidades equivalentes.

Após as refatorações espera-se um sistema com uma maior flexibilidade e modularidade e, consequentemente, com uma maior facilidade de manutenção e evolução, já que tais características são herdadas da programação orientada a aspectos [2].

A linguagem de programação utilizada para as referidas refatorações será AspectJ, uma extensão orientada a aspectos para a linguagem Java. AspectJ representa uma boa alternativa para o desenvolvimento por se tratar de uma linguagem com compatibilidade total com Java e com sua maquina virtual (JVM) [35].

Além disso, serão implementados casos de testes para garantir a integridade das funcionalidades do sistema original.

Ao final do trabalho serão analisados as vantagens e desvantagens do uso da POA, seus impactos serão identificados, além de verificar, a partir de um sistema já implementado, o esforço necessário para tais refatorações. Deste modo, ambos os sistemas servirão como elementos de testbed para pesquisadores utilizarem-nos em seus experimentos e estudos empíricos.

1.3 Estrutura do Documento

Esta seção visa direcionar o leitor para os capítulos que constituem este trabalho, bem como fornecer uma breve explicação dos assuntos abordados em cada capítulo. No capítulo 2, serão apresentados alguns trabalhos e conceitos relacionados com o tema em estudo. Dentre os assuntos abordados estão: estudos empíricos e suas fundamentações; conceitos e características da Programação Orientada a Aspectos; técnicas e aplicabilidade de refatorações na Engenharia de Software. No capítulo 3 são abordados os principais problemas da Programação Orientada a Objetos em que a Programação Orientado a Aspectos busca, através de suas estruturas e funcionalidades, fornecer uma solução sólida de modo a deixar o código mais reusável e com uma maior facilidade de manutenção.

No capítulo 4, o Amadeus é apresentado bem como o ambiente em que ele foi executado, além de propostas de melhoria para o seu código fonte utilizando a Programação

[1]Sistema *open-source* para gestão de aprendizagem desenvolvido em Java

Orientada a Aspectos. Os testes realizados para a comprovação dos resultados também serão exibidos. No capítulo 5, é realizado um resumo de todo os resultados esperados acompanhados dos resultados obtidos com este trabalho. E, por fim, é realizada uma conclusão juntamente com os trabalhos futuros os quais se deseja submetê-lo, afim de aperfeiçoá-lo ou servir como apoio a outros pesquisadores.

2. Trabalhos Relacionados

2.1 Estudos empíricos na engenharia de software

Os métodos empíricos geralmente são tidos como a base dos métodos científicos modernos. Eles defendem que as teorias devem ser baseadas em observações do mundo, em vez da intuição ou fé. Defende, também, a investigação empírica e o raciocínio dedutivo. Segundo John Locke, "Todo o processo do conhecer, do saber e do agir é aprendido pela experiência, pela tentativa e erro".

Para o Empirismo, adquire-se a sabedoria através da percepção do mundo externo, ou então do exame da atividade da mente, que abstrai a realidade exterior e as modifica internamente [36]. Logo, o empirismo é de caráter individualista, pois o conhecimento varia conforme a percepção das pessoas, que é diferente de um indivíduo para o outro. Na Engenharia de Software esse individualismo não pode ocorrer, pois a experiência e os resultados observados por várias pessoas em uma mesma pesquisa é o que vai dar credibilidade ao resultado final.

Estudos empíricos são apenas testes que comparam o que acreditamos ao que vemos. Porém, se os testes forem devidamente construídos, executados e utilizados para apoiar o método científico, os mesmos passam a ter um papel fundamental na ciência moderna [4].

A importância de estudos empíricos na ciência da computação tem sido reconhecida por diversos pesquisadores nos últimos anos. Na engenharia de software, em particular, estudos empíricos são realizados afim de ajudar pesquisadores a avaliar, entender, controlar e melhorar o processo de desenvolvimento de software e seu produto [11].

Experimentação, um tipo de estudo empírico, permite que o conhecimento seja gerado de forma sistemática e controlada, especialmente em áreas guiadas pela intervenção humana [13]. Por esta razão, são muito comuns os experimentos relacionados a fatores sociais e comportamentais, onde o comportamento humano e suas interações são a base para tais experimentos [14]. Porém, a influência humana faz com que experimentos tornem-se muito difíceis de serem planejados e avaliados porque não é possível gerar modelos precisos.

Na Engenharia de Software, um aspecto relevante na experimentação é a necessidade de replicar ou complementar um estudo experimental. Replicação em diferentes contextos é uma característica importante para qualquer tipo de estudo nesta área. A replicação também é

importante para obter-se credibilidade e aprendizagem [7]. Porém, as muitas diferenças entre projetos de software individuais causam uma difícil comparação, afetando o uso dos estudos empíricos [37].

Uma forma de evidenciar os benefícios de uma tecnologia ou metodologia é submetê-la a um ambiente que seja muito parecido com o ambiente real. Porém muitas empresas não estão dispostas a realizar projetos que avaliem uma dada tecnologia. Estudos empíricos são essenciais para preencher esta lacuna e fornecer dados confiáveis, facilitando a transferência de tecnologias. Neste contexto, uma boa alternativa para avaliar tecnologias, num ambiente controlado é o uso de testbeds.

2.1.1 Testbeds

Existe, atualmente, uma grande quantidade de novas tecnologias sendo desenvolvida, porém apenas uma pequena parte delas está sendo utilizada em projetos reais. Isto acontece, principalmente, porque em muitos desses casos, tais tecnologias foram pouco experimentadas e testadas, causando assim, um obstáculo para sua utilização na indústria de software.

Os testbeds são desenvolvidos com o objetivo de avaliar alternativas de tecnologias, além de acelerar sua maturação e transição para uso no mercado. Eles têm sido largamente utilizados em diferentes áreas há muitos anos, porém o uso de testbeds na Engenharia de Software só foi introduzido em 1988, por Robert Holibaugh, J. M. Perry e L. A. Sun [12].

Testbeds são ambientes empíricos nos quais pesquisadores podem planejar e executar estudos com a intenção de avaliar e/ou comparar tecnologias [34]. Testbeds são compostos por um conjunto de artefatos associados a um sistema, além da infraestrutura necessária para a realização dos estudos naquele sistema [15]. Estes artefatos podem incluir a documentação do software, o plano de teste, código-fonte, dados dos resultados e outros artefatos usados em projetos reais. Estes artefatos facilitam a execução dos estudos empíricos, facilitando suas replicações, desde que façam parte do mesmo testbed que, em seguida, poderá ser disponibilizado para outros pesquisadores.

Se os experimentos e suas replicações seguirem o plano de experimentação é possível afirmar, com grande credibilidade, se uma dada aplicação se comporta melhor do que outra. Em muitos casos, pesquisadores utilizam testbeds para ajudar organizações a identificarem se uma dada tecnologia é uma boa alternativa a ser usada no desenvolvimento de seus sistemas. Além disso, também é possível comparar o comportamento de diferentes tecnologias num mesmo cenário, submetendo-as ao mesmo ambiente de desenvolvimento ajudando a determinar qual (ou quais) tecnologias se adaptam melhor às condições propostas [6].

2.2 Programação Orientada a Aspectos

A POA surgiu como um complemento para o paradigma da Programação Orientada a Objetos (POO), onde os requisitos funcionais e não funcionais de uma aplicação podem ser pensados como os interesses (*concerns*) presentes no sistema [38]. O Desenvolvimento de Software Orientado a Aspectos (DSOA) visa aumentar a modularidade dos sistemas, tratando os interesses que não fazem parte dos propósitos básicos do sistema e tornando a estrutura do software mais concisa e, consequentemente, conferindo maior facilidade de manutenção [1]. Tais interesses, referentes aos requisitos não funcionais do sistema, em um sistema utilizando o paradigma POO, encontram-se espalhados por várias classes, por isso são denominados de interesses transversais ou ortogonais (*crosscutting concerns*), pois "atravessam" ou "cortam" várias classes em determinados pontos do código. Entre os exemplos mais comuns desses interesses transversais estão auditoria (registros de logs), persistência, distribuição, controle de concorrência, segurança e tratamento de erros. Um exemplo típico é o problema de logs espalhados pelo sistema, ilustrado pela Figura 2.1. Cada uma das colunas representa umas classe e as respectivas linhas nas mesmas referem-se a chamadas invasivas de logging.

A POA permite que as implementações dos requisitos funcionais (referente ao negócio do sistema) e não-funcionais (interesses não inerentes ao negócio) sejam separadas de forma centralizada. Dessa maneira, o desenvolvimento de software orientado a aspectos busca uma maior separação de interesses (*separation of concerns*) entre os módulos do sistema e, como consequência, busca minimizar a replicação de código e reduzir o acoplamento entre os módulos facilitando sua manutenção. Esses fatores aumentam o potencial de reuso e facilitam a evolução de sistemas complexos [39].

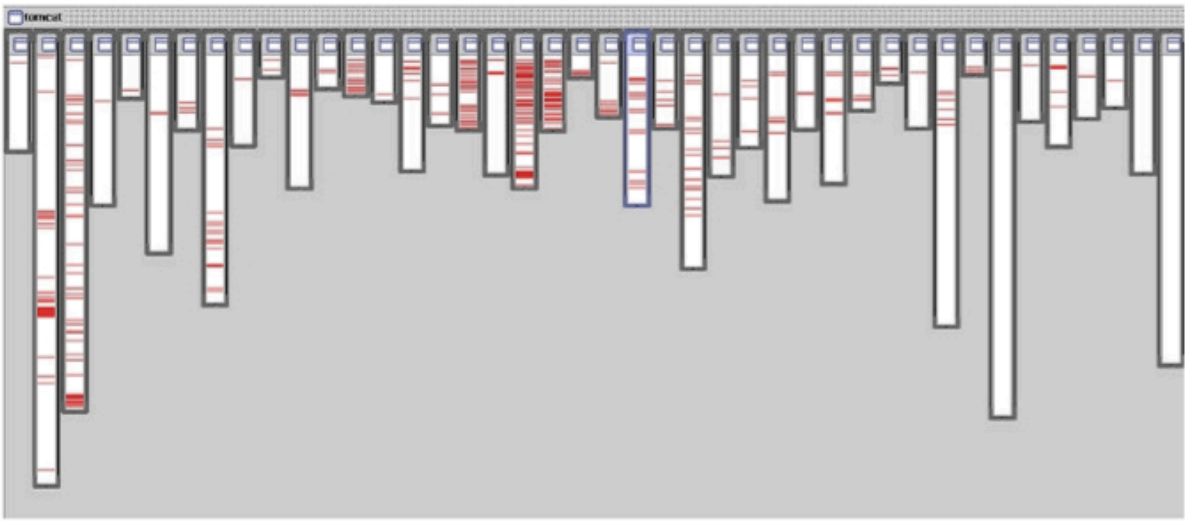

Figura 2.1 - Logs espalhados pelo sistema

2.2.1 Conceitos

Com o intuito de se obter melhores resultados com o uso da POA, as aplicações devem ser planejadas e modeladas seguindo alguns passos. A área de pesquisa conhecida como Modelagem Orientada a Aspectos (MOA) trata da definição de métodos, técnicas, artefatos e processos de modelagem orientados a aspectos. Existem vários trabalhos nessa área, cada uml com uma abordagem diferente e produzindo seu próprio conjunto de artefatos para modelagem modelagem. Porém, há alguns passos básicos que podem ser destacados para a modelagem de um sistema segundo a abordagem orientada a aspectos [40]:

i. **Identificação de interesses (*concerns*)** – representa a identificação dos elementos cruciais ao domínio da aplicação, incluindo os requisitos dos *stakeholders*.
ii. **Definição do Modelo Base** - os interesses não-transversais são identificados. Tais interesses podem ser modelados utilizando as técnicas convencionais de orientação a objetos (OO).
iii. **Definição do Modelo de Aspectos** - onde serão definidos os interesses transversais, ou seja, conceitos que se encontram dispersos por diferentes locais do sistema e que não podem ser adequadamente modelados usando técnicas OO.
iv. **Definições das regras de composição** - define onde e como os interesses transversais (aspectos) irão atuar sobre os elementos do Modelo Base, incluindo a precedência entre aspectos que interceptem um mesmo elemento.
v. **Identificação e resolução de conflitos entre aspectos** - A partir da modelagem, a POA introduz sobre o POO um conjunto de novas abstrações e mecanismos para facilitar o desenvolvimento de software. Assim, a POA complementa a POO por tratar uma nova dimensão para decomposição de responsabilidades no escopo do programa, que modularizam os conceitos transversais: os aspectos.

A implementação de aplicações utilizando POA consiste dos seguintes elementos:

- **Linguagem de programação de propósito geral** - linguagem de programação usada pelo programador para escrever programas que implementam as funcionalidades básicas do sistema (lógica do negócio), ou seja, que implementam o Modelo Base. Neste contexto, componentes são definidos como unidades funcionais da implementação, como por exemplo, objetos, métodos, procedimentos [1]. O código que implementa a lógica do negócio é conhecido como código base e exemplos dessas linguagens são Java, C#, C++.
- **Linguagem de aspectos** - fornece construções básicas para o programador criar as estruturas necessárias para descrever o comportamento dos aspectos (ou seja, implementar o Modelo de Aspectos) e definir os critérios que estabelecem o ponto em que um aspecto será executado. Por exemplo, o programador pode

definir no aspecto critérios que possibilitem a interceptação de um método ou construtor de uma classe. Exemplos de linguagens de aspectos são AspectJ, PHPAspect e AspectC++.

- **Programas de componentes** - Um componente é um módulo que agrega um conjunto de regras de negócio específicas. Programas de componentes são os arquivos de código fonte do programa, desenvolvidos em uma linguagem de programação escolhida, onde estão especificadas as regras de negócio do sistema [1].
- **Programa de aspectos** - São os arquivos de código fonte do programa desenvolvidos na linguagem de aspectos que implementam os interesses transversais do sistema [1].
- **Compositor de aspectos (*aspect weaver*)** - responsável por combinar os programas escritos em linguagem de propósito geral (ou seja, o código base) com os programas escritos em linguagem de aspectos. O processo encarregado de possibilitar ao aspecto interceptar o código base é conhecido como *weaving*, e pode ser estático ou dinâmico. No *weaving* estático, o código a ser injetado no código base é carregado em tempo de compilação. Por outro lado, no *weaving* dinâmico, tal código pode ser adicionado e removido em tempo de execução. [41]. Isso permite que a manutenção do sistema seja modular, pois não envolve a alteração direta no código fonte, bastando alterar o código do aspecto para que o *weaver* possa gerar novamente o programa, e consequentemente obter um novo comportamento do sistema onde o aspecto sofreu a alteração.

Segundo os conceitos da orientação a aspectos, a composição de aspectos com o código base é feita através da definição de pontos de junção *(join points*). Pontos de junção representam pontos bem definidos na execução de um programa que podem ser interceptados por um determinado aspecto como, por exemplo, a chamada ou execução de um método, o acesso a um atributo e a ocorrência de uma exceção. Essas interceptações são definidas de acordo com as regras da linguagem orientada a aspectos utilizada, que definem quais os locais do código fonte que podem ser afetados, alterando o comportamento do programa. Tais regras darão origem aos pontos de atuação (*pointcuts*).

Pontos de atuação (*pointcuts*) são expressões da linguagem para definir os pontos de junção, ou seja, o ponto de atuação torna possível a inserção de comportamento no ponto de junção.

Adendos (*advices*) são constituídos de código a ser injetado na aplicação quando acontece um determinado evento, ou seja, o código definido no *advice* é executado quando o *join point* correspondente ao *pointcut* for capturado. Os *advices* podem ser executados antes,

durante ou depois (*before*, *around*, *after*) da execução do *join point* relacionado.

Um aspecto pode conter métodos além das definições citadas acima, sendo assim, um aspecto deve ser declarado como abstrato sempre que possuir métodos ou *pointcuts* abstratos. Outros elementos presentes na POA são as declarações intertipos (*intertype declarations*). Declarações intertipos são declarações referentes à estrutura de um programa, por exemplo, um aspecto poderia ser utilizado para adicionar novos métodos e atributos a uma classe, ou declarar que uma classe estende uma nova superclasse [41].

Para que o aspecto intercepte o código base é necessário que seja realizado o processo de combinação aspectual, o qual é um processo que antecede a compilação, gerando código intermediário na linguagem de componentes capaz de produzir a operação desejada, ou de permitir a sua realização durante a execução. As classes referentes ao código base não sofrem alterações para suportar POA, isso é feito no momento da combinação entre os componentes e os aspectos, conforme apresentado na Figura 2.2. Dessa forma, o combinador utiliza os aspectos e os componentes para gerar um novo código [41].

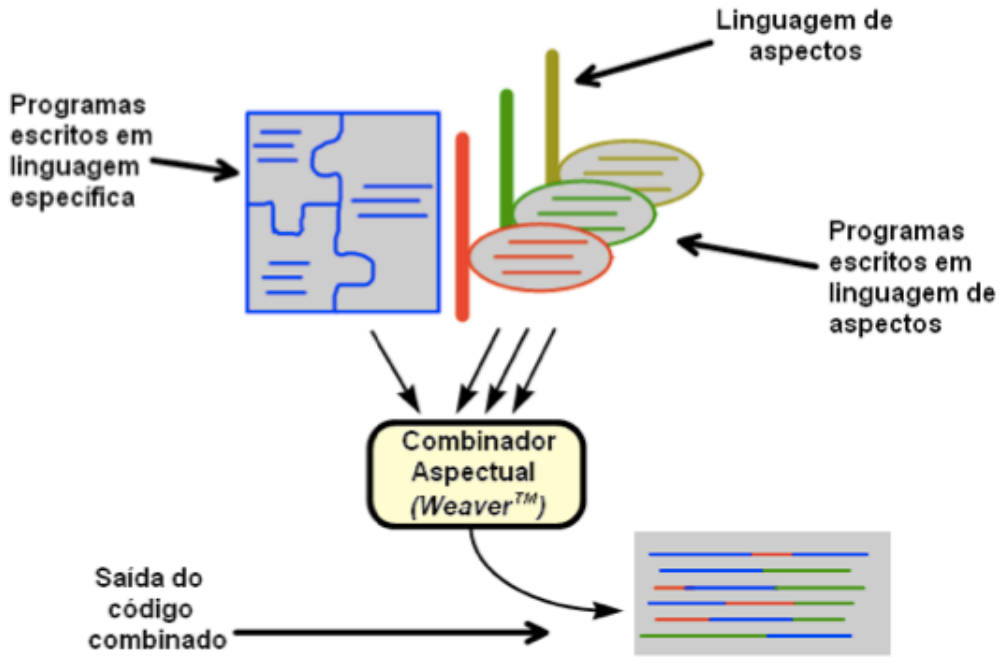

Figura 2.2 - Composição de um sistema baseado em aspectos

2.2.2 AspectJ

AspectJ é uma extensão para a linguagem Java, incorporando à linguagem uma nova unidade de abstração: o aspecto. Como dito anteriormente, além do aspecto existem ainda elementos que caracterizam o paradigma orientado a aspectos, como os *advices, pointcuts e*

join points.

No projeto do AspectJ foram levados em consideração alguns fatores importantes em relação à compatibilidade com a linguagem Java. AspectJ apresenta compatibilidade total com Java, ou seja, todo programa Java puro é um programa AspectJ válido, e todo programa AspectJ pode ser executado em uma JVM (*Java Virtual Machine*) [35]. Em AspectJ, os aspectos são definidos por declarações similares às declarações de uma classe. As declarações de aspectos podem incluir *pointcuts*, *advices*, além de outros tipos de declarações permitidas nas declarações de classes, como métodos e atributos. Ainda é possível utilizar os conceitos de herança presentes na orientação a objetos (permitindo a hierarquização entre aspectos) e o conceito de aspectos abstratos de modo a aumentar o nível de abstração [35].

Os pontos de junção (*join points)* podem representar a invocação e execução de métodos, inicialização de objetos, execução de construtores, tratamento de exceções, acesso e atribuição de atributos, entre outros [42]. A definição de *pointcuts* é feita com a utilização de construtores fornecidos pelo AspectJ denominado designadores de *point cut* (*point cut designator*), conforme apresenta a tabela 2.1.

Tabela 2.1 - Designadores de pointcuts [42]

call(*Assinatura*)	Invocação de método / construtor identificado por *Assinatura*
execution(*Assinatura*)	Execução de método / construtor identificado por *Assinatura*
get(*Assinatura*)	Acesso a atributo identificado por *Assinatura*
set(*Assinatura*)	Atribuição de atributo identificado por *Assinatura*
this(*PadrãoTipo*)	O objeto em execução é instância de *PadrãoTipo*
target(*PadrãoTipo*)	O objeto de destino é instância de *PadrãoTipo*
args(*PadrãoTipo*, ...)	Os argumentos são instâncias de *PadrãoTipo*
within(*PadrãoTipo*)	O código em execução está definido em *PadrãoTipo*

Já os *advices* em AspectJ complementam a implementação do aspecto determinando o comportamento dos *join points* que interceptarão a execução do programa. Os tipos de *advices* são apresentados na tabela 2.2:

Tabela 2.2 - Tipos de advices [42]

before	Executa quando o *join point* é alcançado, mas imediatamente antes da sua computação
after returning	Executa após a computação com sucesso do *joint point*
after throwing	Executa após a computação sem sucesso do *joint point*

after	Executa após a computação do *join point*, em qualquer situação
around	Executa quando o *join point* é alcançado e tem total controle sobre sua computação

A seguir é apresentado um exemplo simples de um programa utilizando AspectJ. Primeiramente, na figura 2.3, declara-se uma classe *HelloWorld,* a qual contém o método *hello()*, que recebe uma *String* como parâmetro e imprime essa *String*. Em seguida, na figura 2.4 declara-se um aspecto, em que na linha 2 é definido um *pointcut* para capturar a execução do *join point* determinado (o método *hello* da classe *HelloWorld*). Na linha 3 declara-se o *advice* (o *advice* contém o código a ser injetado na aplicação quando o *pointcut* é capturado), onde através da palavra-chave *before* é indicado que o código definido no *advice* será executado antes da execução do *join point.* Por outro lado, o uso do *after* no lugar de *before,* faria com que o advice fosse executado após o *join point.*

Por fim, na figura 2.5, tem-se a classe *Main,* responsável por iniciar a execução da aplicação, que terá como saída a *String* "Olá teste".

```
public  class HelloWorld {
    public static void hello (String name) {
        System.out.println(name);
    }
}
```

Figura 2.3 - Declaração da classe HelloWorld

```
public aspect AspectHello {
   pointcut helloWorld() : execution (* HelloWorld.hello(..));
   before : helloWorld() {
       System.out.print("Olá");
   }
}
```

Figura 2.4 - Declaração de um aspecto em AspectJ

```
public class Main {
   public static void main (String args[]) {
       HelloWorld.hello("teste");
   }
}
```

Figura 2.5 - Chamada ao método hello da classe HelloWorld

2.2.3 Problemas da POA

Apesar do DSOA fornecer meios poderosos para uma boa modularização de software podendo atingir muitas partes do sistema sem grandes mudanças, suas estruturas devem ser utilizadas com precaução, pois o uso dos operadores possibilita afetar várias partes do programa, inclusive as partes não desejadas [42]. Alguns dos principais problemas são listados a seguir.

Debugging[2] - apesar do código POA estar separado sintaticamente do código-base, em tempo de execução, eles estão todos juntos. Essa junção de código em tempo de execução pode tornar imprevisível o comportamento do programa caso não fique bem definido o escopo de cada aspecto. Com isso, pode tornar-se difícil a leitura do código do programa dificultando sua depuração.

Captura não intencional de *join points* – um caso bem típico desse problema é quando se define um *pointcut* para um certo padrão de nomenclatura e um programador cria um método ou classe compatível com essa padrão de forma não intencional levando à execução inadvertida do *advice*. Com isso, outro problema pode ser identificado: mudanças no código base podem afetar a semântica dos *pointcuts*. Da mesma forma do exemplo anterior, ao renomear um método ou classe, *join points* podem sem capturados erroneamente.

Todos os programadores devem conhecer o padrão de nomenclatura adotado pelos aspectos para que tais problemas sejam evitados. O ideal seria a existência de uma ferramenta que criasse condições para auxiliar os programadores nesses cenários, fazendo com que tais *advices* ficassem mais visíveis tornando mais fácil de serem identificados. Porém essa questão ainda encontra-se em aberto.

[2] processo de encontrar e reduzir efeitos num software.

2.3 Refatorações

Refatorar é o processo de mudar um programa a fim de melhorar sua estrutura interna e reusabilidade sem alterar seu comportamento original (preservação semântica) [33]. Também pode ser considerada uma técnica para minimizar as chances de introduzir novos bugs, melhorando a qualidade de alguns fatores internos, tal como modularidade, com o intuito de deixar o código mais fácil de entender e facilitar futuras evoluções.

Em sistemas reais, mesmo que o desenvolvedor tente implementar o código da maneira mais coerente desde o começo do desenvolvimento, sempre existirá a necessidade de alguma mudança no software durante o seu ciclo de vida. Seja para tornar o código mais legível, seja para melhorar a sua performance ou até mesmo para deixá-lo mais flexível para futuras implementações ou manutenções. Ao se sentir a necessidade de realizar alguma mudança no software, a abordagem mais comum é a de implementar as novas funcionalidades de forma que demande a menor quantidade de alteração possível no código já existente. Uma vantagem é que isso reduz o risco de introduzir novos bugs ao software.

Apesar de refatorações serem extremamente estudadas para programação procedural e programação orientada a objetos, existem apenas alguns poucos estudos sobre o desenvolvimento de padrões de refatoração, principalmente no que diz respeito à refatoração de programas POO para POA [18].

Além disso, a refatoração orientada a aspectos difere das refatorações tradicionais por seus elementos afetarem muitas áreas ao longo do sistema, tornando mais difícil prever se o comportamento do programa permaneceu inalterado. Inclusive, algumas ferramentas já foram desenvolvidas para verificar se ao fim do processo de refatoração do software algum tipo de alteração semântica foi realizada [19].

Os benefícios da POA tais como modularização, evolutibilidade e a remoção das características negativas da presença dos interesses ortogonais do sistema, junto com a robusteza das linguagens orientadas a aspectos, faz com que o trabalho de reengenharia de sistemas OO seja muito recompensador [31]. Em aspetos, uma das grandes dificuldades do processo de refatoramento é a determinação de *pointcuts* efetivos para interceptar a execução do programa e redirecioná-lo para seu devido *advice*. Tal processo precisa ser realizado de tal forma que preserve o comportamento original.

Como o DSOA introduz novos tipos de estrutura como *advices* e *pointcuts*, a forma de refatorar sistemas de POO para POA é um pouco diferente de outras abordagens. Além de adicionar ao código de aspectos essas novas estruturas, algumas vezes é necessário também alterar o código Java para assegurar que os *join points* serão devidamente capturados. Algumas dessas mudanças é a renomeação de métodos para seguir um certo padrão de

nomenclatura, a extração de métodos para que cada método tenha um comportamento bem definido ou até mesmo a simples transferência do código Java para aspectos.

Existem algumas ferramentas que ajudam os desenvolvedores a refatorar o código de forma segura e eficiente garantindo sua qualidade e manutenabilidade. Alguns estudos propõem ferramentas de geração de código e refatoramento para auxiliar os desenvolvedores a reestruturar programas Orientado a Objetos em programas Orientado a Aspectos. Tais ferramentas aumentam a produtividade do desenvolvimento automatizando algumas tarefas repetitivas. Em [32], por exemplo, é proposta uma ferramenta em AspectJ para ajudar na refatoração de programas em Java.

3. Aplicações da POA

Como dito anteriormente, os exemplos mais comuns dos interesses transversais em que a POA tem interesse de separar do código de negócio afim de melhorar a modularidade do sistema são auditoria (registros de log), persistência, distribuição, controle de concorrência, segurança e tratamento de erros. Através da programação orientada a aspectos, existem várias formas de cada um desses casos serem abordados. Neste seção apresentados alguns conceitos e casos de aplicação da POA em cada uma dessas áreas.

3.1 Log

Sabe-se que um sistema de log muitas vezes é primordial para a depuração de programas no desenvolvimento de aplicações. As principais IDEs do mercado possuem sistemas de depuração muito eficientes, porém o uso de *logging* pode tornar esse trabalho ainda mais ágil, além de poder deixar registrado informações valiosas a respeito da execução do programa.

Algumas das abordagens de implementação mais comuns são registrar a chamada de todos os métodos do sistema (o que seria muito custoso), armazenar os registros de requisições a uma certa classe (uma entidade básica, por exemplo) ou ainda deixar registrado as chamadas aos métodos de uma certa camada, como a camada de fronteira entre a interface e a camada de negócio ou até mesmo a camada de persistência.

Atualmente, há duas APIs de logging em Java que são mais largamente usadas: a *Log4J* e o pacote *java.util.logging*. O Log4J é uma biblioteca open source desenvolvido como subprojeto da Apache Software Foundation's Logging Services Project. Esta biblioteca é muito usada pela comunidade open source, incluindo grande projetos como JBoss e Hibernate.

A arquitetura do Log4J foi construída baseada principalmente em três conceitos: loggers, appenders e layouts. Esses conceitos permitem desenvolvedores registrarem os logs de acordo com seus tipos e prioridades, e controlar onde as mensagens serão exibidas e como

elas serão exibidas quando elas estiverem no local desejado. Loggers são os objetos chamados pela aplicação para iniciar o logging das mensagens. Quando uma mensagem é dada para ser registrada, os loggers geram um evento chamado de LoggingEvent para capturá-la. Então, os loggers enviam os LoggingEvents para os appenders associados. Os appenders enviam a informação contida no LoggingEvent para a saída especificada. Essa saída pode ser, por exemplo, controlada por um ConsoleAppender que registra a mensagem pelo System.out ou por um FileAppender que coloca a mensagem num arquivo de log. Antes de enviar o LoggingEvent para o seu destino, alguns appenders usam layouts para colocar a informação em algum formato específico. Por exemplo, o Log4J possui uma classe chamada XMLLayout que pode ser usada para formatar os LoggingEvents em um XML.

No Log4J, cada LoggingEvent possui um nível que indica sua prioridade. Os níveis padrão do Log4J, organizados por ordem decrescente de prioridade, são: OFF, FATAL, ERROR, WARN, INFO, DEBUG e ALL. Loggers e appenders também possuem níveis e só executaram as requisições de log daqueles eventos que possuem o nível maior ou igual ao seu.

O pacote java.util.logging (JUL), introduzido em 2002 pela Sun com o JDK 1.4, é extremamente parecido com o Log4J e usa quase os mesmos conceitos, apenas renomeando alguns deles. Por exemplo, os appenders são chamados de handlers, os layouts são formatters e os LoggingEvents são LogRecords. O JUL trata os níveis de log do mesmo jeito que o Log4J, porém ele possui nove níveis, ao invés de sete. Portanto, desenvolvedores familiarizados com a biblioteca Log4J pode facilmente entender o JUL, apenas ajustando seu vocabulário.

Porém, apesar de suas grandes semelhanças essas bibliotecas também possuem suas diferenças. Essa diferença pode ser resumida em: “Tudo o que o JUL pode fazer, o Log4J também pode – e ainda mais”. O Log4J possui mais tipos de formatadores de texto, possui mais opções de configuração e é mais flexível na sua implementação.

No que diz respeito a codificação, o registro de log normalmente se dá de uma forma muito simples. Basta inicializar estaticamente o objeto responsável por registrar as mensagens e chamar seus métodos.

```
static Logger logger = Logger.getLogger(LoggingTest.class);
logger.setLevel(LEVEL.INFO);
logger.debug(“Isso não vai aparecer”);
logger.info(“Inicializando...”);
```

Quadro 3.1 - Exemplo de logging utilizando Log4J.

No quadro 3.1, o objeto da classe *Logger* é usado para enviar as mensagens de log. A seguir, o método *setLevel()* é utilizado para ajustar o nível de logging para INFO. A primeira instrução que realmente envia uma mensagem para o log é "Isso nao vai aparecer...". Note que o nível de logging foi ajustado para INFO, que é um nível acima de DEBUG, portanto essa instrução não enviará nada para o log.

Um problema identificado na abordagem utilizando POO é que se faz necessário inserir esse mesmo código em todas as classes e módulos em que se deseja registrar o log, causando o problema chamado de entrelaçamento de código. Percebe-se, portanto, quão trabalhoso isso seria caso fosse necessário fazer o registro de log em dezenas de classes.

Porém, a POA viabiliza uma forma muito mais ágil de viabilizar esse processo. Não é necessário escrever código em todos os locais em que se deseja realizar o log, o aspecto realiza este trabalho automaticamente. Além disso, todo o código responsável pelo logging fica centralizado num só lugar, conseguindo uma maior consistência e modularização.

3.2 Persistência

Num sistema computacional, a persistência de dados diz respeito ao armazenamento não-volátil de dados, como por exemplo, armazenamento de objetos em bases de dados. A persistência de dados é muito comum em qualquer tipo de aplicação desenvolvida hoje em dia, pois é muito importante manter os estados dos objetos para futuras necessidades. Associada a persistência de dados estão também o controle de conexão e de transação, caching de objetos para melhorar a performance e sincronização dos estados dos objetos com a entidade correspondente no banco de dados para garantir a consistência.

No paradigma POO, a implementação mais comum para a camada de persistência é a criação de classes responsáveis pela comunicação com o banco de dados, onde cada classe é responsável por trabalhar com um tipo básico de dados. Muitas vezes é necessário repetir trechos de código nessas classes por elas trabalharem com os mesmos tipos de mecanismos e verificações. Outra costume muito comum é a forma de inicializar a camada de persistência. Os programadores geralmente colocam sua inicialização junto com a inicialização de outros objetos e módulos do sistema. Isso faz com que a tão almejada modularização seja corrompida, pois o entrelaçamento de código de diferentes responsabilidades ...

Em AOP, uma abordagem realizada por [20] é a criação de um aspecto abstrato, o qual é responsável por capturar a inicialização do sistema de forma isolada dos códigos de negócio o programa e, a partir daí, tenta inicializar também a camada de persistência. O termo "tenta" é utilizado porque podem ocorrer falhas no sistema impedindo essa inicialização de ser realizada. Essa falha também foi prevista e tratada pelo código. Com isso, só o aspecto concreto, o que implementa o aspecto abstrato, precisa ser modificado para suportar os

diferentes mecanismos de armazenamento, como banco de dados orientado a objetos ou banco de dados relacionais, por exemplo.

3.3 Distribuição

Um sistema distribuído é uma coleção de computadores independentes que parecem ao usuário como um único sistema único e consistente. Essa definição implica hardware formado por máquinas autônomas e software fornecendo a abstração de uma máquina única [23].

Por ambiente distribuído entende-se um conjunto de processadores interligados por uma rede de interconexão e sem memória compartilhada. A ausência de memória compartilhada obriga a uma interação entre processadores de uma forma distinta do ambiente centralizado: ao invés de variáveis ou arquivos compartilhados utiliza-se troca de mensagens.

O protocolo de distribuição RMI (Remote Method Invocation – Invocação de Métodos Remotos) é uma das formas de abordagem da linguagem Java de prover as funcionalidades a objetos distribuídos, até antes não fornecida pela linguagem. O protocolo RMI é um mecanismo mais sofisticado para a comunicação entre objetos distribuídos Java do que seria uma simples conexão de socket, pois os mecanismos e protocolos por meio dos quais os objetos se comunicam são definidos e padronizados [22]. Assim, é possível acessar um objeto ativo em uma máquina virtual, e que esse objeto virtual possa se comunicar remotamente com objetos de outras máquinas virtuais Java, não importando aonde essas outras máquinas virtuais se encontram.

Protocolos de Distribuição apresentam-se como trechos de implementação que não se referem ao mapeamento de negócio, e que geralmente poluem o código com chamadas e controles repetitivos, inerente às tecnologias de distribuição. Os trechos de implementação referentes a tais protocolos tendem a dificultar a manutenção, prejudicar o entendimento e provocar retrabalhos da troca de tecnologia [21]. Portanto, o tratamento desse espalhamento de código deve ser padronizado com o uso da POA de modo a separar a lógica de negócio da distribuição.

3.4 Controle de Concorrência

Programas concorrentes definem ações que podem ser realizadas simultaneamente. Diferentemente de programas sequenciais, onde só uma atividade é realizada por vez, programas concorrentes podem especificar duas ou mais atividades sequenciais que podem ser executadas concorrentemente como processos paralelos (ou threads) [24, 25].

A programação concorrente está ganhando importância por ser suportada pelas principais linguagens de programação orientadas a objeto, como C# e Java [27][28] e tem sido essencial pelo crescimento do marketing das CPU multi-core [29]. Aplicações concorrentes

têm sido utilizadas principalmente em sistemas que requerem muito processamento – Web Servers, por exemplo. Porém, com o advento dos processadores multi-core, o uso da concorrência tem sido muito comum em aplicações convencionais. Porém, ainda é complexo desenvolver aplicações concorrentes usando linguagens tradicionais. Depurar aplicações concorrentes também é muito complexo principalmente porque sua execução pode ser imprevisível.

Devido aos vários interesses tratados pela programação concorrente, seu código geralmente apresenta os bem conhecidos espalhamento e entrelaçamento de código. Dentre esses interesses estão: quais tarefas podem ser executadas em threads diferentes; a aplicação de políticas que coordene a ordem de execução de tarefas que não podem ser executadas paralelamente por causa de alguma dependência.

Já foram feitos vários estudos com o intuito de melhorar o processo de desenvolvimento de aplicações concorrentes. Inclusive algumas bibliotecas já foram desenvolvidas afim de reduzir o número de linhas de código necessárias para adicionar um comportamento concorrente em aplicações. Porém este tipo de bibliotecas não funcionam com problemas associados a interesses transversais [26].

A Programação Orientada a Aspectos visa trazer à programação concorrente os benefícios da modularização, legibilidade de código e mais independência de desenvolvimento, teste e configuração. Em [26] é feito um estudo com a utilização de composição de padrões utilizando a POA. O uso desses padrões melhora o design de aplicações concorrentes, facilitando o reuso de código e diminuindo a complexidade de manutenção.

3.5 Segurança

Segurança da Informação está relacionada com proteção de um conjunto de dados, no sentido de preservar o valor que possuem para um indivíduo ou uma organização. São características básicas da segurança da informação os atributos de confidencialidade, integridade e disponibilidade, não estando esta segurança restrita somente a sistemas computacionais, informações eletrônicas ou sistemas de armazenamento. O conceito se aplica a todos os aspectos de proteção de informações e dados.

Os principais atributos que, orientam a análise, o planejamento e a implementação da segurança de informações são: Confidencialidade (propriedade que limita o acesso às entidades autorizadas pelo proprietário da informação), Integridade (atribute que assegura que a informação mantenha as características originais estabelecidas pelo proprietário) e Disponibilidade (garante que a informação esteja sempre disponível).

Dentre os mecanismos de segurança mais utilizados estão os mecanismos de controle de acesso. Tais mecanismo incluem sistemas biométricos e firewalls. Com o crescimento da

WEB, está cada vez mais comum a existência de sistemas rodando inteiramente na internet. Tais sistemas, portanto, precisam controlar o acesso do usuário, seja para manter o registro de atividades, seja para permitir acesso a uma determinada área ou funcionalidade do sistema fornecendo privilégios ao usuário. A forma mais comum e simples de desenvolvimento de um sistema de controle de acesso é através da utilização da combinação id/senha fornecida pelo usuário e, caso essa combinação esteja correta, o usuário é autenticado com seus privilégios.

Uma das ações mais comuns no processo de controle de acesso é verificar se certo usuário tem permissão de executar determinada tarefa, como editar, apagar ou visualizar alguma área do sistema, por exemplo. Utilizando Programação Orientada a Objetos, uma das formas mais comuns de se realizar essa verificação é fazer chamadas a alguma classe ou módulo que realiza tal operação para que a execução do programa possa seguir a diante.

Diante disso, deparamos com problemas que a Programação Orientada a Aspectos procura tanto combater: o espalhamento de código de mesmo propósito por várias partes do sistema e o entrelaçamento de código de propósitos diferentes num mesmo local.

Sendo assim, esse cenário representa uma ótima oportunidade para o uso da POA, pois a centralização e o reuso de código são de suma importância em sistemas que tendem a evoluir.

3.6 Tratamento de Exceções

O mecanismo de tratamento de exceções é de grande importância para sistemas baseados na Programação Orientada a Objetos, pois com ele é possível tratar erros e falhas que podem eventualmente ocoorrer durante a execução programa. As formas de tratar exceções são infinitas, ficando a cargo do programador definir qual a mais apropriada para cada caso.

Dentre as formas mais comuns de tratamento de exceções estão: desfazer operações que foram realizadas até aquele ponto do programa, apresentar uma mensagem de aviso ao usuário, registrar o ocorrido num arquivo de log ou até mesmo todas elas ao mesmo tempo.

O tratamento de exceções, por padrão, é um interesse transversal pois é de interesse de todo o sistema. Geralmente ele é implementado com códigos espalhados sobre todos os módulos existentes, porém é possível tratá-lo separadamente do restante dos outros códigos do sistema. Diante disso, se faz necessário utilizar a POA para centralizar todas as regras e estruturas relacionadas ao tratamento de exceções.

4. Estudo de Caso - Amadeus

4.1 Justificativa

O objeto de estudo utilizado neste trabalho foi o projeto Amadeus [3] (sistema open-source para gestão de aprendizagem desenvolvido em Java) que utiliza unicamente programação orientada a objetos (POO).

A escolha deste projeto foi baseada principalmente em quatro fatores:

a) O*pen-source*: Por ser de código aberto, o projeto Amadeus pode ser facilmente encontrado e utilizado por qualquer pessoa que deseja fazê-lo. Além disso, também é possível ao desenvolvedor contribuir livremente com a comunidade compartilhando e difundindo seus conhecimentos e estudos realizados.
b) Linguagem de programação e Orientação a Objetos: O fato do Amadeus utilizar POO e a linguagem de programação *Java* na sua implementação também foi importante para sua escolha, pois além da POO poder proporcionar uma melhor reusabilidade, escalabilidade e manutenabilidade do código se comparada ao paradigma procedural, por exemplo, a linguagem *Java* facilita a refatoração do código para AspectJ, visto que esta é um complemento da primeira.
c) Localização gerencial e administrativa: O projeto Amadeus foi totalmente desenvolvido sob a coordenação do Centro de Informática (CIn) da Universidade Federal de Pernambuco e seus colaboradores locais, o que facilita a comunicação e a troca de ideias e experiências.
d) Importância: o Amadeus representa, hoje, um grande avanço na área de ensino a distância no Brasil, sendo destaque em eventos e revistas. Em seu pouco tempo de vida, ganhou vários prêmios pelo país como o Prêmio Amadeus Partner Gold em 2009 e o Prêmio Pernambuco Inovador em 2011. Além disso, o Projeto Amadeus, ingressou no Portal do Software Público Brasileiro em 2009 , que hospeda projetos considerados estratégicos para o Governo Federal por gerar economias e impacto social.

4.2 Definição do ambiente

O AMADeUs-MM (Agentes Micromundo e Análise do Desenvolvimento no Uso de Instrumentos MultiMídia) é um ambiente virtual de aprendizagem baseado na integração de mídias digitais diversas que visam aproveitar as propriedades destas do ponto de vista da aprendizagem. O AMADeUs prevê a disponibilização de cursos a um conjunto de usuários, que podem assumir diferentes papéis dentro do ambiente, tais como aluno, professor, monitor, dentre outros. As formas de interação entre os usuários do AMADeUs ocorre através da interface web da aplicação, de seus componentes chat, mural, fórum e email.

O projeto Amadeus é composto por três subprojetos: AmadeusLMS, AmadeusGames e AmadeusMobile. Por ser um projeto *open-source*, seu código fonte pode ser acessado livremente e encontra-se em: https://svn.softwarepublico.gov.br/svn/amadeus.

Para que o sistema funcione em sua totalidade algumas aplicações são necessárias. É preciso que a versão de cada aplicação seja seguida à risca para garantir a fidelidade do comportamento do sistema como um todo. As aplicações necessárias são:

- Máquina Virtual Java (JRE) 6 ou superior.
    - Pode ser obtido no próprio site da Oracle: http://www.oracle.com/technetwork/java/javase/downloads/index.html
- Apache Tomcat 6.0.18. Atualmente o Amadeus foi homologado apenas com a versão 6.0.18 do Tomcat e há registros de não funcionamento com versões mais recentes.
    - Pode ser obtido em http://tomcat.apache.org/
- Banco de Dados PostgreSQL 8.3.6.
    - Encontrado em http://www.postgresql.org/download/

Caso o objetivo seja desenvolver e contribuir com o projeto, devem ser utilizados ainda:

- JDK (Java Development Kit) 1.6.
- IDE de desenvolvimento (eclipse, por exemplo).

Para a finalidade deste trabalho, por fazer uso da Programação Orientada a Aspectos com AspectJ, ainda foi necessária a instalação do plugin AJDT (*AspectJ Development Tools*) para eclipse disponível em http://www.eclipse.org/ajdt/.

Não existe na documentação do Amadeus especificações para uma configuração mínima de hardware, porém é recomendável se ter, no mínimo, um processador 800 MHz Intel Pentium III ou equivalente, 512MB de memória e 750MB de espaço livre em disco. Neste trabalho, contudo, foi utilizado o seguinte ambiente:

- Sistema Operacional: Mac OS X versão 10.6.8
- Processador: Intel Core 2 Duo 2.4 GHz
- Memória: 8 GB

4.2.1 Configuração

Após baixar e instalar cada uma das aplicações citadas, se faz necessário realizar algumas configurações.

Para o PostgreSQL, na página do Amadeus no portal do Software Público [8], existem os scripts *amadeuslms_web-v00.95.00.sql* e *amadeuslms_mobile-v00.95.00.sql* que devem ser executados, cada um em sua própria base de dados.

Em seguida, o arquivo *hibernate.cfg.xml* deve ser configurado com os valores do usuário, senha e nome do banco de dados utilizados e, posteriormente, copiado para o diretório do projeto no Tomcat em WEB-INF/classes/. Um exemplo de parte dessa configuração pode ser vista no quadro 4.1.

```
<property name="hibernate.dialect">
        org.hibernate.dialect.PostgreSQLDialect</property>
<property name="hibernate.connection.driver_class">org.postgresql.Driver</property>
<property name="hibernate.connection.url">
        jdbc:postgresql://localhost:5432/amadeus_web</property>
<property name="hibernate.connection.username">postgres</property>
<property name="hibernate.connection.password">password</property>
<property name="hibernate.current_session_context_class">thread</property>
```

Quadro 4.1 - Exemplo de configuração do hibernate.

Finalmente, basta copiar o arquivo *amadeuslms.war* no diretório *webapps* do Tomcat e a aplicação será iniciada em http://localhost:8080/amadeuslms.

Porém, se o objetivo é desenvolver e contribuir com o projeto Amadeus é preciso ainda, depois de baixar o código fonte, adicionar ao *Build Path* do projeto as bibliotecas localizadas em /WebContent/WEB-INF/lib no próprio projeto.

O Manual de instalação e configuração completo, além dos arquivos auxiliares também podem ser encontrados em [8].

4.3 Interesses candidatos à refatorações

Foram identificadas, no projeto Amadeus, algumas oportunidades de melhoria que podem ser contornadas com o auxílio da Programação Orientada a Aspectos. Nesta seção serão apresentadas tais melhorias juntamente com as soluções propostas.

4.3.1 Log

Foi identificado que o sistema Amadeus ainda possui uma lacuna a ser preenchida no que diz respeito ao registro de logs. Primeiramente, foi observado que dentre as dezenas de classes do projeto, apenas três delas realizam algum tipo de log. Além disso, essas três classes não utilizam um padrão, deixando cada implementação diferente uma da outra.

```
public class HibernateSessionRequestFilter implements Filter {

    private static Log log = LogFactory.getLog(HibernateSessionRequestFilter.class);

    private SessionFactory sf;

    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain)
            throws IOException, ServletException {

        try {
            log.debug("Starting a database transaction");
```

Figura 4.1 - Implementação de log no Amadeus

```
DateFormat formatter = new SimpleDateFormat("dd/MM/yy");
Date start = null;
Date finish = null;

try {
    start = (Date) formatter.parse(myForm.getString("startEvaluation"));
    finish = (Date) formatter.parse(myForm.getString("finishEvaluation"));
} catch (ParseException e) {
    log.error(e.getMessage());
};
```

Figura 4.2 - Implementação de log no Amadeus em outra classe

Na figura 4.1 é utilizado um objeto da classe *org.apache.common.logging.Log* para o registro de logs. Já na figura 4.2, como a classe em questão herda de *org.apache.struts.actions.DispatchAction,* observamos o uso do objeto *org.apache.struts.actions.DispatchAction.log* já existente na classe herdada. Faz-se extremamente necessário, portanto, a existência de uma estrutura que auxilie o registro de logs não só nas classes mencionadas, mas também em qualquer local do sistema que seja pertinente de forma flexível. É proposta a implementação de um aspecto que realiza a captura automática dos pontos de interesse do sistema e realiza o registro de log das assinaturas dos métodos executados.

```
public aspect AspectMain {

    Logger _logger = Logger.getLogger("trace");

    AspectMain() {
        _logger.setLevel(Level.ALL);
    }

    pointcut tracePoints(): !within(br.ufpe.cin.amadeus.aspects.AspectMain) &&
        !within(tests.TestCaseAspects) &&
        !call(*.new(..)) && !execution(*.new(..)) &&
        !initialization(*.new(..)) &&
        !staticinitialization(*);

    before() : tracePoints() {
        if (_logger.isEnabledFor(Level.INFO)) {
            _logger.log(Level.INFO, "========= " + thisJoinPoint + " ==========");
            _logger.log(Level.INFO, "Static join point information:");
            printStaticJoinPointInfo(thisJoinPointStaticPart);
            _logger.log(Level.INFO, "Enclosing join point information:");
            printStaticJoinPointInfo(thisEnclosingJoinPointStaticPart);
        }
    }

    private void printStaticJoinPointInfo(StaticPart joinPointStaticPart) {
        _logger.log(Level.INFO, "Signature: " + joinPointStaticPart.getSignature());
        _logger.log(Level.INFO, " Kind: " + joinPointStaticPart.getKind());
        SourceLocation sl = joinPointStaticPart.getSourceLocation();
        _logger.log(Level.INFO, "Source location: " + sl.getFileName() + ":" + sl.getLine());
    }
```

Figura 4.3 - Proposta de implementação para logging no Amadeus

Na solução proposta, exibida na figura 4.3, é feito o registro de log (utilizando o Log4J) de todos os métodos a serem executados com exceção dos métodos da classe de testes, os construtores dos objetos e inicializadores. Vale salientar que apesar de o nível de log ter sido colocado diretamente no código, seria totalmente possível criar um arquivo de configuração para que esse nível fosse facilmente modificado, dando mais flexibilidade ao código.

Como exemplo, uma parte do log obtido com a execução do método *existEmail(String)* é mostrado a seguir:

```
09:41:56,359  INFO trace:36 - Source location: Facade.java:73
09:41:56,359  INFO trace:24 - ========== execution(boolean br.ufpe.cin.amadeus.amadeus_web.facade.Facade.existEmail(String)) ==========
09:41:56,360  INFO trace:25 - Static join point information:
09:41:56,360  INFO trace:33 - Signature: boolean br.ufpe.cin.amadeus.amadeus_web.facade.Facade.existEmail(String)
09:41:56,360  INFO trace:34 -  Kind: method-execution
09:41:56,360  INFO trace:36 - Source location: Facade.java:237
09:41:56,360  INFO trace:27 - Enclosing join point information:
09:41:56,360  INFO trace:33 - Signature: boolean br.ufpe.cin.amadeus.amadeus_web.facade.Facade.existEmail(String)
09:41:56,360  INFO trace:34 -  Kind: method-execution
09:41:56,360  INFO trace:36 - Source location: Facade.java:237
09:41:56,361  INFO trace:24 - ========== get(Controller br.ufpe.cin.amadeus.amadeus_web.facade.Facade.controller) ==========
09:41:56,361  INFO trace:25 - Static join point information:
09:41:56,361  INFO trace:33 - Signature: Controller br.ufpe.cin.amadeus.amadeus_web.facade.Facade.controller
09:41:56,361  INFO trace:34 -  Kind: field-get
09:41:56,361  INFO trace:36 - Source location: Facade.java:238
09:41:56,361  INFO trace:27 - Enclosing join point information:
09:41:56,361  INFO trace:33 - Signature: boolean br.ufpe.cin.amadeus.amadeus_web.facade.Facade.existEmail(String)
09:41:56,361  INFO trace:34 -  Kind: method-execution
09:41:56,362  INFO trace:36 - Source location: Facade.java:237
09:41:56,362  INFO trace:24 - ========== call(boolean br.ufpe.cin.amadeus.amadeus_web.facade.Controller.existEmail(String)) ==========
09:41:56,362  INFO trace:25 - Static join point information:
09:41:56,362  INFO trace:33 - Signature: boolean br.ufpe.cin.amadeus.amadeus_web.facade.Controller.existEmail(String)
09:41:56,362  INFO trace:34 -  Kind: method-call
09:41:56,362  INFO trace:36 - Source location: Facade.java:238
09:41:56,362  INFO trace:27 - Enclosing join point information:
09:41:56,362  INFO trace:33 - Signature: boolean br.ufpe.cin.amadeus.amadeus_web.facade.Facade.existEmail(String)
09:41:56,362  INFO trace:34 -  Kind: method-execution
09:41:56,363  INFO trace:36 - Source location: Facade.java:237
09:41:56,363  INFO trace:24 - ========== execution(boolean br.ufpe.cin.amadeus.amadeus_web.facade.Controller.existEmail(String)) ==========
09:41:56,363  INFO trace:25 - Static join point information:
09:41:56,363  INFO trace:33 - Signature: boolean br.ufpe.cin.amadeus.amadeus_web.facade.Controller.existEmail(String)
```

Figura 4.4 - Resultado do logging

Foram utilizados os valores padrão do Log4J, porém é possível alterar o arquivo de configurações (log4j.properties) para mudar, por exemplo, o local onde será salvo o log, o formato do arquivo, layout do texto entre outras opções.

4.3.2 Controle de Acesso

O Amadeus utiliza um método estático *isLoggedUser(HttpServletRequest)* por todo o sistema afim de verificar a existência de algum usuário logado juntamente com seus privilégios para que seja permitida ou proibida a execução de algumas ações do usuário. Essa verificação, feita em muitos lugares espalhados pelo sistema de forma desordenada, é realizada consultando-se a sessão do objeto *request*, onde são armazenados algumas informações relativas ao usuário. Além disso, quando se faz necessário checar as permissões do usuário para a execução de uma certa ação (execução do método *answerForumActivity*, por exemplo), ainda é preciso consultar todas as permissões desse usuário no banco de dados, podendo sobrecarregar o sistema.

```
295    public ActionForward answerForumActivity(ActionMapping mapping, ActionForm form,
296            HttpServletRequest request, HttpServletResponse response) throws Exception {
297
298        ActionForward forward = null;
299
300        if(SystemActions.isLoggedUser(request)) {
301            DynaActionForm myForm = (DynaActionForm) form;
302
303            int idForum = Integer.parseInt(request.getParameter("idForum"));
304            Forum forum = facade.getForumById(idForum);
305
306            AccessInfo loggedUser = (AccessInfo) request.getSession().getAttribute("user");
307            AccessInfo user = facade.searchUserById(loggedUser.getId());
308
```

Figura 4.5 - Método com uso de controle de acesso no Amadeus

Percebe-se na figura 4.5 que na linha 300 é verificado a existência de um usuário logado e na linha 307, com o id do usuário logado, é requisitado no banco de dados todos os privilégios desse usuário. É nítido o equívoco dessa implementação, pois num cenário em que o usuário precise executar várias ações, serão realizadas várias requisições ao banco de dados de forma desnecessária. Além disso, existe ainda o método *showViewWelcome*, utilizado para enviar o usuário para a página inicial caso ele não tenha permissão de execução de alguma tarefa. Tal método chamado cada vez que o método *isLoggedUser* retorna um valor falso.

Dessa forma, a POA pode auxiliar o desenvolvedor no que diz respeito ao controle de acesso fazendo com que a verificação seja feita de forma automatizada para que não seja necessário realizar chamadas aos métodos de forma explícita e desnecessária. Com isso, conseguimos separar bem as responsabilidades das implementações (código de controle de acesso separado das ações do usuário) e centralizar o controle de chamadas ao método em questão. A solução proposta é apresentada a seguir:

```
pointcut accessControl(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) :
    call(public org.apache.struts.action.ActionForward
            br.ufpe.cin.amadeus.amadeus_web.struts.action.content_management.evaluation.EvaluationActions.*
                (ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)) &&
    call(public org.apache.struts.action.ActionForward
            br.ufpe.cin.amadeus.amadeus_web.struts.action.content_management.forum.ForumActions.*
                (ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)) &&
    call(public org.apache.struts.action.ActionForward
            br.ufpe.cin.amadeus.amadeus_web.struts.action.content_management.ModuleActions.*
                (ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)) &&
    call(public org.apache.struts.action.ActionForward
            br.ufpe.cin.amadeus.amadeus_web.struts.action.manager.ManagerUserActions.*
                (ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)) &&
    call(public org.apache.struts.action.ActionForward
            br.ufpe.cin.amadeus.amadeus_web.struts.action.SystemActions.*
                (ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)) &&
    !call(public org.apache.struts.action.ActionForward
            br.ufpe.cin.amadeus.amadeus_web.struts.action.SystemActions.showViewWelcome
                (ActionMapping, ActionForm, HttpServletRequest, HttpServletResponse)) &&
    !call(*.new(..)) && !execution(*.new(..)) &&
    args(mapping, form, request, response);

ActionForward around(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) :
            accessControl(mapping, form, request, response) {

    ActionForward forward = null;

    if(SystemActions.isLoggedUser(request)) {
        forward =  proceed(mapping, form, request, response);
    } else {
        String content = Facade.getInstance().getMostPopularKeywwordsPreparedAsHTML();

        request.setAttribute("content", content);
        request.setAttribute("webSettings", SystemActions.webSettings);

        forward = mapping.findForward("fShowViewWelcome");
    }
    return forward;
}
```

Figura 4.6 - Proposta de implementação do controle de acesso

Nesta implementação utilizando-se AspectJ, exibida na figura 4.6, são utilizados pointcuts para capturar todos os métodos que requerem a verificação de validação de usuário. Caso o usuário esteja logado, o referido método será executado, caso contrário será executado o método *mapping.findForward("fShowViewWelcome")*, que redirecionará o usuário para a página inicial do sistema. Além disso, foi mantido o padrão de codificação utilizado pelo projeto.

Uma ressalva, porém, precisa ser feita. Essa codificação pode ser realizada desta forma porque os métodos que necessitam realizar a verificação já possuem um padrão (mesmo modificador de acesso, tipo de retorno, classe e parâmetros). Contudo, se tal padrão não existisse haveria de ser feito algumas mudanças no código como, por exemplo, adicionar algum prefixo ou sufixo em todos os métodos para que os pointcuts os capturassem corretamente. Uma outra alternativa seria o uso de *Annotations*. Poderíamos colocar uma *Annotation* em cima dos métodos que precisassem ser verificados. Na figura 4.7 vemos um exemplo de um aspecto utilizando *Annotation*:

```
pointcut accessControlAnnotation():
    call(@br.ufpe.cin.amadeus.amadeus_web.facade.AccessControlRequired * *.*(..));
```

Figura 4.7 - Uso de Annotations

Dessa forma, todo método que possuir a *Annotation* AccessControlRequired seria capturado por esse *pointcut*.

4.3.3 Exceções

E, por fim, foram detectadas oportunidades de melhorias na forma de tratamento de exceções do Amadeus. É de suma importância a existência de uma estrutura que facilite a localização do ponto de falha do sistema. No Amadeus, quando isso acontece é apresentada apenas uma tela-padrão de erro para o usuário.

A solução proposta foi a de registrar o log de erro, como feito em 4.3.1. Na figura 4.8 é apresentada uma solução utilizando log4J para realizar o registro de log dos *join points* capturados. Uma outra solução também avaliada foi que o ideal seria que o usuário soubesse qual foi o erro encontrado pelo sistema. De forma que a mensagem de falha de execução fosse exibida junto com a tela de erro. Porém, ainda não foi encontrada uma forma de implementar tal funcionalidade.

```
pointcut exceptionLog() : call(* *.*(..)) && !within(AspectMain);

after() throwing(Throwable t): exceptionLog() {

    String message = " Class : " + t.getClass()
        + "\n " + "Message :" + t.getMessage();
    _loggerEx.error(getStackMessage(message, t));
}

private static String getStackMessage(String message,
                                      Throwable exception)
{
      StringBuilder sb = new StringBuilder();
      sb.append(" [ " );
      sb.append(exception.getClass().getName());
      sb.append(" ] ");
      sb.append(message);
      return sb.toString();
  }
```

Figura 4.8 - Proposta de implementação do aspecto

Vale salientar que as soluções aqui propostas são apenas formas de se resolver o problema mesmo que parcialmente. Tais estudos foram realizados com o intuito de mostrar que com o uso da POA existem várias formas de melhorar a qualidade do sistema e a sua manutenabilidade.

4.4 Testes

Alguns casos de teste foram criados para garantir que o funcionamento do código esteja de acordo com o planejado e que a refatoração utilizando POA não inseriu novos bugs ao sistema.

No caso das funcionalidades relativas ao log do sistema, o caso de teste foi realizar chamadas a alguns métodos do sistema e verificar a existência de um arquivo contendo todo o log desejado. Portanto, não foi possível a comparação exata com o código já existente, pois o sistema original continha falhas na sua implementação, cabendo à refatoração fazer algumas melhorias.

Já no que diz respeito ao controle de acesso do usuário, foram realizados testes para que fosse verificado automaticamente se esta funcionalidade se comportava como o esperado. Foi utilizado o StrutsTestCase for JUnit [43] – uma extensão do JUnit TestCase que provê facilidades para testar códigos baseados no framework Struts. Através dessa extensão é possível inserir atributos na requisição do usuário, realizar verificações na sessão, entres outras funcionalidades.

E, por fim, foi criado um caso de teste para a verificação do comportamento da funcionalidade tocante a captura de exceções. Foi forçada a existência de um erro em meio a execução do programa para que o seu tratamento fosse observado e, consequentemente, avaliado.

5. Resultados

Nesta seção os resultados esperados serão apresentados enquanto que será feita uma avaliação dos resultados obtidos com este trabalho.

5.1 Resultados Esperados

Os principais resultados esperados neste trabalho são listados a seguir:

- Um estudo quantitativo sobre algumas técnicas da POA (a serem escolhidas).
- Concepção de casos de testes de algumas técnicas estudadas.
- Refatoração de um sistema resultando num sistema equivalente (mesmo comportamento) ao sistema original, porém com algumas técnicas da POA aplicadas a

ele. Estes sistemas auxiliará a construção de um testbed para manutenção de software orientado a aspectos.

5.2 Resultados Obtidos

Após a realização deste trabalho foram alcançados todos objetivos previamente esperados. Através de estudos e pesquisas bibliográficas foram detectados os benefícios associados ao uso da Programação Orientada a Aspectos, sendo eles fatores decisivos para sua a adoção. Além disso, foram realizados testes automatizados que comprovassem o comportamento das funcionalidades descritas. E, consequentemente, com a refatoração do sistema de POO para POA, foi concebido um artefato para auxiliar a construção de um testbed para manutenção de software orientado a aspectos.

6. Conclusão e Trabalhos Futuros

A implementação da POA se comparada ao paradigma orientado a objetos é mais atrativa, pois resulta num código com uma maior modularização evitando o entrelaçamento de código com diferentes propósitos e menor espalhamento de código pelo sistema causado principalmente pelos interesses transversais. Além disso, a possibilidade de reutilização dos módulos, torna o uso da POA uma alternativa promissora para o tratamento dos requisitos ortogonais e uma maior flexibilidade para a evolução de sistemas.

Em relação a linguagem AspectJ, por ela ser uma extensão orientada a aspectos da linguagem Java, apresenta total compatibilidade com sua JVM. A linguagem possui construtores poderosos que podem alcançar várias partes do sistema devendo-se, portanto, ser usada com precaução. A maior vantagem de sua utilização é a possibilidade de implementar funcionalidades de forma separada dos outros códigos do sistema, deixando o código muito mais legível. Além disso, é muito fácil inserir e remover os aspectos do processo de recomposição.

Contudo, a refatoração do sistema baseado em POO para POA pode ser considerado uma tarefa árdua, pois exige do programador um bom conhecimento sobre o código do sistema. Por isso, a ausência desse conhecimento pode acarretar a inserção de novos bugs ao sistema. Com o intuito de que isso não acontecesse os testes desenvolvidos foram satisfatórios e seu comportamento ocorreu como esperado.

Uma dificuldade encontrada foi a compatibilidade do servidor de aplicações Tomcat com a versão do AspectJ, pois algumas vezes foram detectados erros devido a uma falha de comunicação entre eles. Sendo assim, os testes ficaram restritos aos testes automatizados porque o sistema não apresentava estabilidade enquanto era executado no servidor. Espera-se que com este trabalho pesquisadores tomem conhecimento dos benefícios da POA e possam utilizar o conteúdo aqui desenvolvido para suas pesquisas e estudos empíricos. Além disso, é de grande importância que o código desenvolvido possa ser utilizado também pela comunidade do Amadeus, ajudando seus colaboradores com os benefícios da POA aqui avaliados.

Referências

[1] KICZALES, G.; IRWIN, J.; LAMPING, J.; LOINGTIER, J.; LOPES, C. V; MAEDA, C.; MENDHEKAR, A. Aspect-Oriented Programming. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland. Springer-Verlag,1997.

[2] Definição de um Testbed para o Desenvolvimento de Software, Sérgio Soares – Clube da Revista – (NUTES – HC – UFPE) – Abril 2010.

[3] http://amadeus.cin.ufpe.br/

[4] Métodos Empíricos na Engenharia de Software - Danniel Rocha, Francisco Célio, Gelter Thadeu, Germano de Oliveira, Zoralia Brito

[6] Architecture and Application of autonomous robotic software engineering technology testbed (SETT), Alexander K. Lam. A Dissertation Presented to the FACULTY OF THE GRADUATE SCHOOL UNIVERSITY OF SOUTHERN CALIFORNIA In Partial Fulfillment of the Requirements for the Degree DOCTOR OF PHIILOSOPHY (COMPUTER SCIENCE), December 2007.

[7] Replicação de estudos empíricos em Engenharia de Software, Emerson Silas Dória. São Carlos. Maio, 2011.

[8] http://www.softwarepublico.gov.br/dotlrn/clubs/amadeus/file-storage/index?folder_id=27751418, acessado em outubro de 2011.

[9] Using the methodology of Aspect-Oriented Development in the local market, Recife, June of 2007.

[11] V R Basili, R W Selby, and D H Hutchens. Experimentation in software engineering. IEEE Trans. Softw. Eng., 12(7):733–743, 1986.

[12] Robert Holibaugh, J. M. Perry, and L. A. Sun. Phase i testbed description: Require- ments and selection guidelines. Technical report, Software Engineering Institute, Carnegie-Mellon University, September 1988.

[13] Claes Wohlin, Per Runeson, Martin H¨ost, Magnus C. Ohlsson, Bj¨oorn Regnell, and Anders Wessl´en. Experimentation in software engineering: an introduction. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[14] Colin Robson. A Resource for Social Scientists and Practitioners-Researchers. Wiley- Blackwell, 2 edition, 2002.

[15] Mikael Lindvall, Ioana Rus, Forrest Shull, Marvin Zelkowitz, Paolo Donzelli, Atif Memon, Victor Basili, Patricia Costa, Roseanne Tvedt, Lorin Hochstein, Sima As- gari, Chris Ackermann, and Dan Pech. An evolutionary testbed for software tech- nology evaluation. Innovations in Systems and Software Engineering, 1:3–11, April 2005.

[16] Samuel T. Redwine, Jr. and William E. Riddle. Software technology maturation.

In ICSE ’85: Proceedings of the 8th international conference on Software engineering, pages 189–200, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[17] Isanio Lopes Araujo Santos. Uma Abordagem baseada em Aspectos e Composição Dinâmica para a Construção de Aplicações Adaptativas Cientes ao Contexto. Universidade Federal do Rio Grande do Norte, 2008.

[18] Masanori Iwamoto, Jianjun Zhao. Refactoring Aspect-Oriented Programs

[19] Charles Zhang and Hans-Arno Jacobsen, Julie Waterhouse, Adrian Colyer. Aspect Refactoring Verifier∗

[20] Sérgio Soares, Eduardo Laureano, Paulo Borba. Implementing Distribution and Persistence Aspects with AspectJ, Informatics Center, Federal University of Pernambuco Recife, Pernambuco, Brazil.

[21]Antônio Maria Pereira de Resende; Claudiney Calixto da Silva. Programação Orientada a Aspectos em Java, Desenvolvimento de Software Orientado a Aspectos, 2005.

[22] LEMAY, L.; CADENHEAD, R. Aprenda em 21 Dias Java 2. Campus, São Paulo, 1999.

[23] TANENBAUM, Andrew S. Sistemas Operacionais Modernos – Rio de Janeiro: Editora Prentice-Hall do Brasil Ltda, 1995.

[24] Andrews, G., Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley, 2000.

[25] Andrews, G., Schneider F., Concepts and Notations for Concurrent Programming, ACM Computing Surveys, 1983

[26] Carlos Augusto da Silva Cunha. Reusable Aspect-Oriented Implementations of Concurrency Patterns and Mechanisms, Setembro de 2006.

[27] Lea, D., Concurrent Programming in Java, Second edition, Addison-Wesley, 1999.

[28] Oaks, S., Wong, H., Java Threads, 3rd edition, O’Reilly 2004.

[29] Sutter, Herb, The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software, Dr. Dobb’s Journal, 30(3), March 2005.

[30] D. Binkley, M. Ceccato, M. Harman, P. Tonella, Automated Pointcut Extraction, 2005.

[31] Miguel P. Monteiro, João M. Fernandes, Aspect-oriented Refactoring of Java Programs

[32] Paulo Borba, Sergio Soares, Refactoring and Code Generation Tools for AspectJ, Informatics Center, 2002.

[33] Fowler, M. (with contributions by Beck, K., Opdyke, W., Roberts, D.) (1999). Refactoring – Improving the Design of Existing Code. Addison Wesley.

[34] Emanoel Francisco Sp´sito Barreiros, A systematic Mapping Study on Software Engineering Testbed, Centro de Informática, Recife, Fevereiro de 2011.

[35] KICZALES, G., et al. An Overview of AspectJ. In: PROCEEDINGS OF THE 15th EUROPEAN CONFERENCE ON OBJECT- ORIENTED PROGRAMMING (ECOOP). 2001. Budaspest, Hungary. London, United Kingdom: Spring-Verlag. Lecture Notes In Computer Science (LNCS).

[36] CHAUÍ, M. *Convite a Filosofia*. São Paulo: Ática, 2003.

[37] BASILI, V., *Editorial.* Empirical Software Engineering Journal, 1996. 1(2).

[38] PACIOS, S. F. Uma abordagem orientada a aspectos para o desenvolvimento de linhas de produto de software. São Carlos, 2006. Dissertação (Mestrado em Ciência da Computação e Matemática Computacional). Instituto de Ciências Matemáticas e Computação – ICMC, Universidade de São Paulo (USP).

[39] FIGUEIREDO, E., et al. Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method. In: PROCEEDINGS OF THE 9TH WORKSHOP ON QUANTITATIVE APPROACHES IN OBJECT- ORIENTED SOFTWARE ENGINEERING (QAOOSE) CO-LOCATED WITH ECOOP'05. 2005. Glasgow, United Kingdom.

[40] SCHAUERHUBER, S.;RETSCHITZEGGER, W. Towards a Common Reference Architecture for Aspect-Oriented Modeling. In: 8TH INT. WORKSHOP ON ASPECT ORIENTED MODELING (AOM). 2006. Bonn, Germany.

[41] WINCK, D. V.; GOETTEN JUNIOR, V. AspectJ: Programação Orientada a Aspectos com Java. 1. ed. São Paulo, SP, Brasil: Novatec Editora. 2006.

[42] SOARES, S.; BORBA, P. AspectJ – Programação orientada a aspectos em Java. 2002. Disponível em: <http://www.cin.ufpe.br/~scbs/artigos/AspectJ_SBLP2002.pdf>. Acesso em: 25 novembro 2011.

[43] http://strutstestcase.sourceforge.net/