

**LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO DO RN
Bacharelado em Sistema de Informação**

FERNANDO DA CRUZ LOPES

**DESENVOLVENDO UMA ARQUITETURA ORIENTADA A
ASPECTOS E APLICANDO-A NO DESENVOLVIMENTO DE
SISTEMAS**

FERNANDO DA CRUZ LOPES

**DESENVOLVENDO UMA ARQUITETURA ORIENTADA A
ASPECTOS E APLICANDO-A NO DESENVOLVIMENTO DE
SISTEMAS**

Monografia apresentada ao Curso de Bacharelado em Sistema de Informação, da FARN – Faculdade Natalense para o Desenvolvimento do RN, como requisito para a obtenção do título de Bacharel em Sistema de Informação.

Orientador: Rômulo Fagundes (Esp.)

AGRADECIMENTOS

A Deus, que me deu força e conhecimento para concluir este trabalho.

Aos meus pais e minha esposa Cristiane pela paciência e noites de sono mal dormidas.

Aos amigos de trabalho que, na medida do possível, esclareceram dúvidas e apontaram erros que foram corrigidos.

A todos os professores que conseguiram passar de forma eficiente uma parte de seus conhecimentos os quais foram de fundamental importância para a conclusão deste trabalho.

A todos aqueles que direta ou indiretamente contribuíram para que este trabalho fosse realizado.

... sobre a programação
... técnicas, através de
... é orientação a sistemas, que
... parâmetros. Todo este trabalho
... trabalhar com a biblioteca Aspet,
... além de ajudar nos IDs de
... também influencia.

... da página de ... 2 ... 2 ...

Uma mente que se abre a uma nova ideia jamais voltará ao seu tamanho original.
Albert Einstein

RESUMO

Este trabalho de conclusão de curso apresentará uma visão geral sobre a programação orientada a objetos e alguns problema que ela não consegue resolver facilmente, entrando em cena a programação orientada a aspectos. Iremos falar sobre o que é orientação a aspectos, quais os problemas que ela se propõem a resolver e seus conceitos fundamentais. Todo este trabalho será embasado na linguagem de programação JAVA e por isto iremos trabalhar com a biblioteca AspectJ que nos dá um suporte completo para utilização do aspecto, além de plugins para IDEs de desenvolvimento e uma integração com outras tecnologias que também utilizaremos.

Palavra-chave: POA, aspectos, programação orientada a aspectos, aspectj

ABSTRACT

This work of completing the course will present an overview of object-oriented programming and some problems that it can not solve, coming on the scene the aspect-oriented programming. We'll talk about what is mentoring aspects, what problems it purports to solve, and its fundamental concepts. All this work is grounded in the Java programming language and therefore we will work with the AspectJ library that gives a full support for the use of aspect, as well as plugins for IDEs development and integration with other technologies that we also use.

Keywords: AOP, aspects, aspect-oriented programming , aspectj

LISTA DE FIGURAS

Figura 1 - Classe PessoaDao com os métodos de persistência.....	16
Figura 2 - Classe PessoaDao com métodos de persistência auditados.....	17
Figura 3 - Anatomia de um aspecto.....	25
Figura 4 - Aspecto com anotação.....	26
Figura 5 - Diagrama de classe do padrão Observer.....	26
Figura 6 - Diagrama de compilação de interesses sistêmicos.....	31
Figura 7 - Processo de weaving da POA.....	31
Figura 8 - As três fases do processo de weaving do aspecto.....	32
Figura 9 - Interesses transversais espalhados pelo sistema.....	40
Figura 10 - Estrutura simples de um arquivo POM.....	43
Figura 11 - Declaração das dependências no arquivo POM.....	44
Figura 12 - Declaração de um dependencyManagement no arquivo POM.....	45
Figura 13 - Utilizando properties no arquivo POM.....	45
Figura 14 - Arquitetura invisível.....	47
Figura 15 - Arquitetura anêmica.....	48
Figura 16 - Arquitetura intermediária.....	49
Figura 17 - Arquitetura madura.....	50
Figura 18 - Diagrama das classes de negócio.....	53
Figura 19 - Diagrama das classes e interfaces que darão suporte às entidades.....	54
Figura 20 - Diagrama das classes e interface do pacote persistence query.....	55
Figura 21 - Estrutura de pacotes da arquitetura.....	57
Figura 22 - Aspectos com as regras de pacotes.....	58
Figura 23 - Erro em tempo de compilação gerado pelo aspecto.....	58
Figura 24 - Declaração de dependência no arquivo POM.....	59
Figura 25 - Adicionando um plugin ao maven.....	59
Figura 26 - Erros de compilação gerados pelas regras dos aspectos.....	60
Figura 27 - Erro de compilação ao tentar compilar utilizando os comando do maven.....	60

LISTA DE CÓDIGOS

Código 1 - Estrutura de um pointcut.....	21
Código 2 - Exemplo de um advice do tipo before.....	23
Código 3 - Exemplo de um advice do tipo after.....	23
Código 4 - Exemplo de um advice do tipo around.....	24
Código 5 - Exemplo de utilização de um advice.....	24
Código 6 - Exemplo de utilização de um pointcut juntamente com um advice.....	24
Código 7 - Aspecto que simula o padrão de projeto observer.....	27
Código 8 - Implementando o padrão observer com aspectos.....	28
Código 9 - Código exemplo de uma declaração de um aspecto.....	33
Código 10 - Declaração de um aspecto explicitando o tipo de associação.....	33
Código 11 - Declaração de um aspecto com associação do tipo perthis.....	34
Código 12 - Declaração de um aspecto com associação do tipo pertarget.....	34
Código 13 - Declaração de um aspecto com associação do tipo percfow.....	34
Código 14 - Declaração de um aspecto com associação do tipo percfowbelow.....	34
Código 15 - Método lançando um exceção.....	35
Código 16 - Método com tratamento de exceção.....	35
Código 17 - Método lançando exceção para camadas superiores.....	36
Código 18 - Aspecto que captura e trata exceções.....	36
Código 19 - Exemplo de métodos transacionais.....	37
Código 20 - Aspecto que transaciona operações.....	38
Código 21 - Classe ClienteDAO.....	39
Código 22 - Classe ClienteDAO com a implementação de auditoria.....	39
Código 23 - Aspecto que implementa auditoria.....	41

LISTA DE ABREVIATURAS E SIGLAS

TI	Tecnologia da Informação
OO	Orientação a Objeto
POA	Programação Orientada a Aspectos
POO	Programação Orientada a Objeto
IDE	Integrated Development Environment
XML	Extensible Markup Language
JRS	Java Specification Requests
JDK	Java Development Kit
JVM	Java Virtual Machine
DAO	Data Access Object
POM	Project Object Model
MVC	Model View Controller
CRUD	Create, Read, Update e Delete
JPQL	Java Persistence Query Language
CDI	Contexts and Dependency Injection
JSF	Java Server Faces
JAR	Java Archive

LISTA DE TABELAS

Tabela 1 - Caracteres especiais da linguagem Aspectos.....	21
Tabela 2 - Expressões lógicas da linguagem Aspectos.....	21
Tabela 3 - Pointcuts pode tipo de ponto de junção da linguagem Aspectos.....	21
Tabela 4 - Tipos de ponto de junção.....	22
Tabela 5 - Advice after e seus tipos.....	23
Tabela 6 - Tipos de weaving da POA.....	32

SUMÁRIO

1. INTRODUÇÃO.....	13
2. OBJETIVOS.....	13
2.1. OBJETIVOS GERAIS.....	13
2.2. OBJETIVOS ESPECÍFICOS.....	13
3. REFERENCIAL TEÓRICO.....	14
3.1. PROGRAMAÇÃO ORIENTADA A OBJETO.....	14
3.1.1. O que é programação orientada a objeto.....	14
3.1.1.1. objetos e classes.....	15
3.1.1.2. estado e comportamento.....	15
3.1.2. Onde a orientação a objeto falha.....	16
3.1.2.1. separação de interesses.....	18
3.2. REQUISITOS.....	18
3.2.1. Requisitos funcionais.....	19
3.2.2. Requisitos não funcionais.....	19
3.3. PROGRAÇÃO ORIENTADA A ASPECTOS.....	19
3.3.1. Introdução.....	20
3.3.2. Conceitos.....	20
3.3.2.1. ponto de junção (join point).....	20
3.3.2.2. ponto de atuação (pointcut).....	21
3.3.2.3. adentos (advice).....	23
3.3.2.4. anatomia do aspecto.....	25
3.3.3. Aspectos: com anotação X sem anotação.....	25
3.3.4. Aspectos e padrões de projetos.....	26
3.4. ASPECTJ.....	29
3.4.1. Crosscutting elements.....	29
3.4.1.1. aspects.....	30
3.4.1.2. introduction.....	30
3.4.1.3. compile-time declaration.....	30
3.4.2. O compilador.....	31
3.4.2.1. weaver.....	32
3.5. ASSOCIAÇÕES DE ASPECTOS.....	33
3.5.1. Associações por objeto.....	34
3.5.2. Associações por fluxo de controle.....	35
3.6. ONDE USAR ASPECTOS.....	35
3.6.2. Restrições arquiteturais.....	37
3.6.3. Transação e persistência de dados.....	38
3.6.4. Auditoria.....	39
3.7. MAVEN.....	42
3.7.1. Fase do build.....	43
3.7.2. Project object model (pom).....	43
3.7.3. Dependências.....	44
3.7.4. Dependency Management.....	45
3.7.5. Properties.....	46
3.8. ARQUITETURA DE SOFTWARE.....	46
3.8.1. O que é uma arquitetura de software.....	47
3.8.2. Maturidade arquitetural.....	47
3.8.2.1. arquitetura de software “invisível”.....	47
3.8.2.2. arquitetura de software “anêmica”.....	48
3.8.2.3. arquitetura de software “intermediária”.....	49

3.8.2.4. arquitetura de software "madura".....	50
4. ESTUDO DE CASO	51
4.1. PROPOSTA DE ARQUITETURA.....	51
4.1.1. Processo da arquitetura de software.....	51
4.1.2. Estrutura arquitetural.....	52
4.1.2. Aspectos arquiteturais.....	57
4.2. UTILIZANDO A ARQUITETURA NO DESENVOLVIMENTO DE UM SISTEMA.....	58
4.2.1. Configuração maven.....	59
4.2.2. Restrições na programação em camada.....	59
5. CONCLUSÃO	61
REFERÊNCIAS	62

1. INTRODUÇÃO

Com o passar do tempo e a evolução da área de Tecnologia da Informação (TI), o desenvolvimento de algumas aplicações vem se tornando mais complexo e por consequência, o desenvolvimento orientado a objeto (OO) vem sofrendo para elaborar soluções sem infringir seus próprios conceitos. Outro problema inerente da OO é o fato das classes terem características essencialmente estáticas o que dificulta, em uma possível mudança de requisito, a alteração no sistema podendo causar um profundo impacto na aplicação.

Geralmente, em OO, ficamos focados apenas nos interesses da solução onde uma ou mais classes conseguem encapsular todos os códigos referentes a uma determinada característica do sistema. Isso funciona bem em muitos casos, mas quando os requisitos mudam os problemas começam a aparecer. Por exemplo, a necessidade de programar uma auditoria levaria a códigos duplicados em boa parte do sistema.

Diante desses problemas, o desenvolvimento da orientação a aspectos (OA) emerge, resolvendo a maioria dos problemas que a OO não atende sem quebrar suas próprias regras. Este trabalho de conclusão de curso focará na programação orientada a aspectos (POA), criando uma mini-arquitetura que iremos nos embasar para desenvolver qualquer tipo aplicação, sempre seguindo as regras que foram definidas na arquitetura e nos aspectos nela definidos.

3. OBJETIVOS

3.1. OBJETIVOS GERAIS

Mostrar como a programação orientada a aspectos pode ajudar no desenvolvimento de um sistema, criando aspectos a nível de arquitetura de software, tendo suas regras aplicadas tanto na arquitetura quanto no sistema que utiliza esta arquitetura.

3.2. OBJETIVOS ESPECÍFICOS

- Criação de uma arquitetura de software com funções básicas;
- Desenvolvimento de Aspectos para restrições arquiteturais;
- Integração de aspectos com outras tecnologias;
- Aplicação do aspectos na implementação de sistemas.

3. REFERENCIAL TEÓRICO

Ao longo do tempo diversos paradigmas de programação surgiram sempre com o objetivo de melhorar a modularidade dos programas. Na programação estrutural os programas eram separados em módulos chamados procedimentos, favorecendo a criação de bibliotecas que poderão ser utilizadas por outras aplicações. O paradigma de orientação a objeto apresentou uma estrutura chamada classe, com métodos e atributos os quais podem representar os estados e comportamentos de uma aplicação.

Programas orientados a objetos favorecem certa decomposição do sistema em classes, às vezes referida como decomposição primária (KICZALES; MEZINI, 2005). No entanto, alguns aspectos comuns a diversos sistemas são especialmente difíceis de modularizar usando POO, e tendem a se espalhar por diversas classes. Esses aspectos são denominados interesses transversais (crosscutting concerns). Exemplos típicos incluem logging, autorização, persistência, transações, caching e tolerância a falhas (OSSHER; TARR, 1998). Tais interesses podem ser facilmente implementados utilizando um outro paradigma de programação, a orientação a aspectos.

A programação orientada a aspectos é um paradigma de programação que permite que os interesses transversais sejam implementados em módulos independentes denominados aspectos (KICZALES et al., 1997). Para isso os aspectos definem, de maneira declarativa, alterações na estrutura e no comportamento de classes.

3.1. PROGRAMAÇÃO ORIENTADA A OBJETO

É bem mais fácil programar um sistema e explicá-lo ao cliente se este simula as situações do mundo real. Ao longo dos anos, a programação estruturada já não atendia a essa especificação, falhando ao fornecer mecanismos para abstração eficaz para esse tipo de situação.

Este problema se agravou ainda mais com a popularização do computador, onde muita empresa de médio e grande porte resolveu se informatizar gerando uma demanda muito grande por desenvolvimento de softwares para gerenciar alguns processos operacionais. Diante destas dificuldades, estava se formando o cenário para o surgimento da Orientação a Objeto.

3.1.1. O que é programação orientada a objeto

A OO surgiu para contornar os problemas de abstração que a programação estrutura não

consegue resolver, onde se simula situações do mundo real facilmente, controlando características, comportamentos e estados dos objetos envolvidos no sistema. Ao se fazer tal controle, acabamos por conhecer mais a fundo todo o software, abordando as funções dos objetos em um nível conceitual.

Com a OO, transportou-se o estado do sistema para dentro dos objetos, porém, fazendo algumas restrições de acesso, introduzindo o conceito de visibilidade, onde os alguns métodos ou atributos ficam inacessíveis diretamente, não sendo possível assim acesso ao seu estado. Com essas características trazidas pela OO, temos alguns benefícios como: reuso de objetos, maior modularização do sistema, simplificação do código e por consequência desses benefícios, redução dos custos de manutenção [WINCK, Diogo Vinícius], grande problema que a demanda aumentada de software gera nas empresas desenvolvedoras.

3.1.1.1. objetos e classes

A grosso modo, podemos dizer que as classes é como estruturamos os nossos objetos, criando os atributos, método e estendendo outras classes mais abstratas ou ainda implementando interfaces. Alguns desses conceitos não serão enfatizados neste trabalho, pois não faz parte do escopo principal.

Quando estruturamos uma classe podemos dizer que ela ainda não tem vida e para dar vida a essa classe devemos instanciá-la, criando assim o conceito de instância. Logo, um objeto nada mais é do que a instância de uma classe, onde fica armazenados todos os comportamentos e o estado do domínio que aquela classe representa.

3.1.1.2. estado e comportamento

Segundo Diogo Vinícius Winck, os objetos armazenam informações referentes àquele momento em que está sendo utilizado. Essas informações ficam armazenadas nos atributos das classes e, quando essas informações são acessadas em um dado momento, dizemos que estamos vendo o estado do objeto. Toda a vez que se alterar uma característica do objeto (atributo), por consequência alteramos o seu estado.

Comportamento seria a ação que um objeto executa quando um atributo é alterado ou quando é estimulado por uma mensagem. As ações do comportamento são dadas pelos procedimentos que os métodos executando. Tudo interligando, os métodos acessam os atributos, podendo manipulá-los e, por consequência, alterando o estado do objeto.

3.1.2. Onde a orientação a objeto falha

A OO chegou como uma promessa para resolver uma crise que se instalava no desenvolvimento de software. Com a inovação tecnológica muitas empresas resolveram informatizar partes de seus processos e, com o aumento na demanda por softwares, os desenvolvedores perceberam que os métodos tradicionais de desenvolvimento não conseguiriam dar conta de toda essa demanda. O paradigma OO, por possuir um conjunto de abstrações (atributos, classes, objetos, etc), chegava a ser apontada como a linguagem que viria a resolver todos os problemas de reusabilidade, evitando todo um esforço duplicado no desenvolvimento.

“A OO possibilita um nível maior de abstração quando comparada às metodologias anteriores, permitindo uma melhor representação dos elementos que compõem a solução de um problema. A orientação a objeto introduziu conceitos encapsulamento, herança e polimorfismo, permitindo criar uma hierarquia de objetos que modelam um conjunto comum de comportamentos [WINCK, Diogo Vinícius - pag. 30]”

Com a OO, sempre que vamos programar uma solução de um problema, programamos representações dos elementos e conceitos relacionados ao problema que estamos atacando, criando assim o conceito de domínio. “O Domínio é o espaço onde reside a solução de um problema. [WINCK, Diogo Vinícius - pag. 30]”.

Com esses conceitos que a OO nos apresentou, ficou muito mais eficaz representar os elementos que compõem o domínio do problema, e assim, resolver o problema com mais rapidez, mas isso não é tão verdadeiro quando começamos a perceber que parte de nossa aplicação começam a ter outros tipos de interesses, não mais apenas interesses no domínio do problema e sim interesses transversais. Elementos inerentes a solução começam a surgir, onde estes começam a ter papéis quase que exclusivamente de suporte à aplicação, não fazendo parte do agrupamento original que compõem o domínio.

Para exemplificar este tipo de problema, podemos imaginar uma classe responsável por manter a entidade Pessoa:

```

4 public class PessoaDao {
5     protected EntityManager entityManager;
6
7     public void atualizar(Pessoa pessoa){
8         entityManager.merge(pessoa);
9         entityManager.flush();
10    }
11
12    public void salvar(Pessoa pessoa){
13        entityManager.persist(pessoa);
14        entityManager.flush();
15    }
16
17    public void remover(Pessoa pessoa){
18        entityManager.remove(pessoa);
19        entityManager.flush();
20    }
21 }

```

Figura 1 - Classe PessoaDao com os métodos de persistência

Fonte: Resultado da pesquisa do autor.

Uma classe simples onde o requisito levantado era simplesmente manter o objeto pessoa persistido. Os problemas começam a aparecer quando os requisitos inicialmente definidos recebem alterações e agora se faz necessário auditar todas as ações que a classe pessoa executa o que nos faz rever todo o código.

```

4 public class PessoaDao {
5     protected EntityManager entityManager;
6
7     private Auditoria aud = Auditoria.getLoadClass(PessoaDao.class);
8
9     public void atualizar(Pessoa pessoa){
10        entityManager.merge(pessoa);
11        entityManager.flush();
12        aud.gravaLog("Atualizado com sucesso!");
13    }
14
15    public void salvar(Pessoa pessoa){
16        entityManager.persist(pessoa);
17        entityManager.flush();
18        aud.gravaLog("Gravado com sucesso!");
19    }
20
21    public void remover(Pessoa pessoa){
22        entityManager.remove(pessoa);
23        entityManager.flush();
24        aud.gravaLog("Removido com sucesso!");
25    }
26 }

```

Figura 2 - Classe PessoaDao com métodos de persistência auditados

Fonte: Resultado da pesquisa do autor.

Como podemos claramente perceber, os interesses adicionais são forçados a fazer parte de outro contexto, sendo inserido em outros interesses apenas para dar apoio ao interesse principal, quebrando as regras da própria OO (códigos duplicados em todas as entidades serão auditados). Esses problemas poderiam ser contornados utilizando-se de extensões da orientação a objeto, como

a programação orientada a aspectos (POA).

3.1.2.1. separação de interesses

Um dos conceitos centrais da POA é o conceito de interesses que nada mais são do que as características relevantes de uma aplicação. Separamos essas características geralmente para modularizar nossa aplicação, seguindo os preceitos da OO a fim de evitar códigos duplicados.

A orientação a objeto tende a separar apenas alguns interesses das soluções, geralmente, os interesses funcionais que são formados pelos aspectos relacionados ao domínio da aplicação. Embora estes tipos de interesses serem de grande importância para a aplicação, existe outro tipo de interesse que tem como característica dar suporte à aplicação, estes são conhecidos como interesses sistêmicos ou interesses transversais.

Afinal de contas, o que são esses interesses de suporte à aplicação? Imagine você programar sua classe e implementando aqueles métodos que pode gerar uma exceção caso alguma coisa não ocorra de acordo. Você terá duas alternativas: uma você terá que alterar a assinatura do método, escrevendo a exceção com a palavra reservada `throws`; outra será, sempre que for necessário, abrir os blocos `try/catch` para tratar as exceções. Todos esses trechos de código para tratamento de exceções podem considerá-los como códigos de suporte à aplicação.

Vários outros exemplos podem ser citados como sendo codificação de interesses transversais, como coordenação de múltiplos objetos, a própria persistência e auditoria (`log`). Todos esses interesses, se não programados corretamente, podem gerar o que chamamos de `Scattering Code` (código espalhado) e `Tangled Code` (código emaranhado), é justamente neste ponto onde a programação orientada a objeto começa a falhar, sendo preciso utilizar alternativas como a orientação a aspectos para corrigir estas falhas.

3.2. REQUISITOS

Para iniciar a programação de uma rotina onde iremos atacar determinado problema, precisamos ter de antemão informações de regras de negócio e detalhes do problema para seguirmos o melhor caminho no desenvolvimento da solução. Essas informações são levantadas junto aos clientes e são conhecidas como requisitos do sistema.

Segundo Cysneiros (2001, p. 16) um requisito “é a condição necessária para a obtenção de certo objetivo”. No caso de um software, requisitos são as necessidades do cliente, as características funcionais e não funcionais que o sistema precisa apresentar (MARTINS, 2007).

3.2.1. Requisitos funcionais

Requisitos funcionais são os que definem o comportamento da aplicação, o que ela pode ou não fazer e como reagir a determinados estímulos de entrada (mensagem). Geralmente esses requisitos são descritos de uma forma bem clara, em uma linguagem informal e simples de ser compreendida. São exemplos de requisitos funcionais:

- O sistema não permitirá ao usuário acesso ao menu do sistema sem autenticação
- O sistema permitirá ao usuário cadastrar entrada e saída de produto no estoque
- O sistema não permitirá o cadastro de saída do produto sem estoque

3.2.2. Requisitos não funcionais

São requisitos que declaram restrições, ou atributos de qualidade para um software e/ou para o processo de desenvolvimento deste sistema (Cysneiros 2001, p. 22).

São compostos por características que fazem parte do desenvolvimento do software. Incluem restrições de tempo, restrições no processo de desenvolvimento, padrões, e qualidades globais de um software, como confiabilidade, manutenibilidade, desempenho, portabilidade, suporte, entre outros. Nem sempre dizem respeito ao sistema como um todo, mas sim ao processo em que ele será desenvolvido (CYSNEIROS, 2001; MARTINS, 2007; PAIM, 2003).

Alguns exemplos de requisitos não funcionais:

- O sistema deverá ser compatível com o sistema operacional Windows e Linux
- O sistema deverá possuir um banco de dados compatível com a plataforma a ser utilizada
- O sistema deverá ser igualmente acessado pelos browser Mozilla Firefox, Chrome e Internet Explorer

3.3. PROGRAMAÇÃO ORIENTADA A ASPECTOS

Como já descrito neste trabalho em capítulos anteriores, os paradigmas atuais (como a POO) não atende às necessidades de uma implementação dos requisitos de um sistema completo sem quebra algumas de suas próprias regras. É neste ponto em que a programação orientada a aspectos surge e ganha forças ao promover a separação avançada dos interesses transversais.

Gregor Kiczales, um dos criados da POA, sugere que “para um paradigma de programação conseguir uma ampla aceitação, ele necessita ser expressivo, eficiente, intuitivo, compatível e

possuir uma boa ferramenta que o suporte". A POA já apresenta essas características como veremos a partir de agora.

3.3.1. Introdução

A POA surgiu nos laboratórios da Xerox, por volta de 1997 com o objetivo de construir uma abordagem que fosse um conjunto não necessariamente homogêneo, ou seja, poderia estar completamente separada a aplicação base (central), mais ainda assim dar suporte a esta aplicação base. O objetivo desta abordagem seria tratar as características sistêmicas, também conhecidas como transversais, do comportamento programa. A esta abordagem dá-se o nome de meta-programação, Goetten e Winck (2006).

A preocupação central que incentivou a criação deste paradigma foi o fato da implementação dos interesses sistêmicos gerar uma dependência entre a parte base e a transversal, dificultando uma possível manutenção na parte central do sistema.

3.3.2. Conceitos

Quando falamos de programação orientada a aspectos, três conceitos deste paradigma se destacam: os pontos de junção (join points), pontos de atuação (pointcut) e os adentros (advice).

3.3.2.1. ponto de junção (join point)

Ponto de junção são os pontos que podem ser identificados e afetados pelo aspectos durante a execução do programa. A nossa programação do aspectos pode ser associada a esses pontos de junção e executar uma rotina antes, depois ou ao invés da rotina original.

São alguns pontos de junção conhecidos:

- **Métodos:** podemos programar um aspecto para atuar na chamada ou execução de um método;
- **Construtores:** assim como nos métodos, podemos programar um aspecto para agir durante a chamada de um construtor (*new*) e/ou durante a execução do construtor;
- **Acesso a campos:** aspectos podem atuar nas propriedades (dados) de uma classe de dois modos, um apenas como ponto de leitura sem modificar os dados e outro com escrita, podendo modificar valores;
- **Tratamento de exceções:** um ponto de junção de tratamento de exceções é o interior de um

bloco *catch* que 'captura' algum tipo de exceção;

- **Inicialização de classes:** são pontos de junção que representam o instante em que uma classe é carregada na memória e seus dados estáticos são inicializados.

Há outros tipos de pontos de junção menos usados: inicialização de objetos, pré-inicialização de objetos, e execução de *advice*.

3.3.2.2. ponto de atuação (pointcut)

Ponto de atuação é uma programação na linguagem do aspectos onde conseguimos selecionar ou agrupar um ou mais pontos de junção da minha classe. Seria a sintaxe que escrevemos para selecionar os ponto de uma classe que receberá alguma ação do aspecto. Segue um exemplo de sintaxe básica de um ponto de atuação:

Código 1 - Estrutura de um pointcut

```
public pointcut nome() : call (void Pessoa.print())
```

Explicando o exemplo acima: a palavra *public* já é conhecida dos programadores JAVA e se aplica o mesmo conceito em aspecto, restrição de acesso, e como no JAVA, o pointcut pode ser *public*, *private* e *protected*; a palavra *pointcut* denota que estamos declarando um ponto de atuação, também conhecido como ponto de corte; todo o pointcut tem um nome que pode ou não receber parâmetros; depois dos dois pontos (:) obrigatórios vem o tipo dos pontos de junção agrupados pelo pointcut - nesse caso temos um pointcut do tipo *call*, que indica uma chamada de método; finalmente vem a assinatura do pointcut, uma especificação dos pontos de junção aos quais o pointcut se refere [NELSON, Torsten 2005].

Como o pointcut é a programação que junta os join points, é necessário o uso de alguns caracteres especiais para compor a minha expressão de seleção:

Caracteres	Significado
*	Qualquer sequência de caracteres não contendo pontos
..	Qualquer sequência de caracteres, inclusive contendo pontos
+	Qualquer subclasse de uma classe

Tabela 1 - Caracteres especiais da linguagem Aspectos

Um pointcut pode conter várias assinaturas ligadas através de operadores lógicos. Os operadores são:

Operador	Significado	Exemplo	Interpretação do texto
!	Negação	!Pessoa	Qualquer classe exceto Pessoa
	"ou" lógico	Funcionario Cliente	Classe Funcionario ou classe Cliente
&&	"e" lógico	Cloneable && Runnable	Classes que implementam ambas as interfaces Cloneable e Runnable

Tabela 2 - Expressões lógicas da linguagem Aspectos

Já vimos neste trabalho os pointcuts do tipo *call*, que se refere a chamadas de métodos. Além desse, existem outros tipos, listados a seguir com suas sintaxes:

Categoria de ponto de junção	Sintaxe do pointcut
Chamada de método ou construtor	call(AssinaturaDoMetodo)
Execução de método ou construtor	execution(AssinaturaDoMetodo)
Inicialização de classe	staticinitialization(Classe)
Leitura de dado de classe	get(AssinaturaDoCampo)
Escrita de dado de classe	set(AssinaturaDoCampo)
Tratamento de exceção	handler(Exceção)

Tabela 3 - Pointcuts pode tipo de ponto de junção da linguagem Aspectos

Há dois tipos de pointcuts que se referem a um conjunto de pontos de junção que ocorrem em um **fluxo de controle**. Um fluxo de controle consiste de todos os comandos que são executados entre a entrada em um método e a saída deste, inclusive considerando comandos em outros métodos chamados pelo primeiro [NELSON, Torsten].

Os dois pointcuts são *cflow()* e *cflowbelow()*. A única diferença entre eles é que *cflow()* inclui entre os pontos de junção resultantes o próprio ponto de junção dado como parâmetro, enquanto *cflowbelow()* inclui apenas os pontos de junção que estão "abaixo" do ponto dado [NELSON, Torsten 2005].

A tabela 4 mostra um exemplo de cada tipo:

Categoria de ponto de junção	Sintaxe do pointcut
Chamada de método ou construtor	call(AssinaturaDoMetodo)
Execução de método ou construtor	execution(AssinaturaDoMetodo)
Inicialização de classe	staticinitialization(Classe)
Leitura de dado de classe	get(AssinaturaDoCampo)
Escrita de dado de classe	set(AssinaturaDoCampo)
Tratamento de exceção	handler(Exceção)

Tabela 4 - Tipos de ponto de junção

3.2.3.3. adentos (advice)

Um adento nada mais é do que o comportamento do aspecto, o que irá fazer quando executado. Em termos mais formais, o advice designa a *semântica comportamental* do aspecto. Todo advice está associado a um pointcut, que define pontos de junção [NELSON, Torsten]. Há três tipos de advice:

- *Before*: executando antes do ponto de junção;
- *After*: executado depois do ponto de junção;
- *Around*: executado “ao redor” do ponto de junção; esse tipo de advice serve para substituir a execução do ponto de junção pela execução do advice, ou executar parte do advice antes do ponto e junção e outra parte depois [NELSON, Torsten 2005].

Todos os tipos de advice são definidos de forma semelhante:

Código 2 - Exemplo de um advice do tipo before

```
before():                                     // declaração do advice
ponto_de_corte() {                           // ponto de corte associado
    System.out.println("teste");             // corpo do advice
    System.out.println("tchau");
}
```

O tipo *before* é o mais simples, ele apenas executa antes do ponto de junção. Uma única observação a ser feita é que, o adento levantando uma exceção, o ponto de junção não será executado.

O advice do tipo *after* executa após o ponto de junção. Existem três variações desse advice, que dependem de como terminou a execução do ponto de junção: se terminou normalmente, com uma exceção, ou de qualquer forma.

<i>after()</i> : ponto_de_corte()	executa depois do ponto de junção, independente de como ele retornou
<i>after() returning</i> : ponto_de_corte()	executa depois do ponto de junção se ele tiver terminado normalmente (sem exceção)
<i>after() throwing</i> : ponto_de_corte()	executa depois do ponto de junção somente se este tiver saído com uma exceção

Tabela 5 - Advice after e seus tipos

Tanto a forma *returning* quanto a *throwing* podem definir um parâmetro que conterá o valor retornado ou a exceção jogada, para ser usada pelo aspecto. Por exemplo:

Código 3 - Exemplo de um advice do tipo after

```
after() returning(String s) : call(* Pessoa.getNome()) {
    System.out.println("O metodo retornou o string " + s);
}
```

Já o *advice around* é executado substituindo o ponto de junção. No entanto, o ponto de junção pode ser executado de dentro do corpo do advice usando-se o comando *proceed()*. A chamada a *proceed()* retorna o mesmo tipo retornado pelo ponto de junção substituído. Além disso, se o ponto de junção é um método que espera receber argumentos, estes devem ser passados no *proceed()*. Veremos como passar esses argumentos na seção seguinte.

Todo *advice around* deve declarar um tipo a ser retornado. O tipo deve ser o mesmo tipo dos pontos de junção associados a ele. Por exemplo, se o advice está associado à chamada de um método que retorna um inteiro, ele deve retornar um inteiro. Todos os pontos de junção de um *advice around* devem retornar um tipo compatível com o tipo retornado pelo advice. Por exemplo:

Código 4 - Exemplo de um advice do tipo around

```
String around() : call (* *.toString()) {
    String s = proceed();
    return s.toLowerCase();
}
```

Se os pontos de junção retornam tipos diferentes, pode-se contornar isso declarando que o advice retorna *Object*. O advice cuidará para que o tipo correto seja retornado ao chamador, mesmo que o tipo seja um tipo primitivo.

Da mesma forma que tipos de retorno, o advice deve declarar que joga (throws) exceções jogadas pelo método capturado.

O aspecto pode usar informações do ponto de junção capturado. Essa informação faz parte do contexto do ponto de junção. É dever do ponto de corte (pointcut) expor o contexto desejado para que o advice possa usá-lo. Por exemplo, suponha que queiramos imprimir uma mensagem quando inserimos um ingrediente a uma receita, detalhando a operação efetuada:

Código 5 - Exemplo de utilização de um advice

```
before(Receita r, Ingrediente i) :
    call(void adicionaIngrediente(Ingrediente)) &&
    target(r) &&
    args(i) {
        System.out.println("Inserindo ingrediente"
            + i + " na receita "
            + r.getNome() );
    }
}
```

O pointcut *target* captura o objeto que é o alvo da chamada de método (o *call* especificado). O pointcut *args* captura os argumentos passados como parâmetros (no caso, um objeto da classe *Ingrediente*). Pointcuts do tipo *this* também servem para capturar contexto. No caso, o *this* captura o objeto onde está o ponto de junção (o *call*, nesse caso). Pointcuts já podem ser definidos com os parâmetros desejados. Poderíamos escrever o exemplo anterior da seguinte forma:

Código 6 - Exemplo de utilização de um pointcut juntamente com um advice

```
pointcut insercao(Receita r, Ingrediente i) :
    call(void adicionaIngrediente(Ingrediente)) &&
    target(r) && args(i);

before(Receita r, Ingrediente i) : insercao(r, i) {
    System.out.println("Inserindo ingrediente"
        + i + " na receita " + r.getNome() );
}
```

3.3.2.4. anatomia do aspecto

Diante das características até agora mostradas sobre o aspecto, as ideias de como é seu funcionamento e como cada conceito se relacionam ficam mais claros, podendo assim representar um aspecto como ilustrado na imagem abaixo:

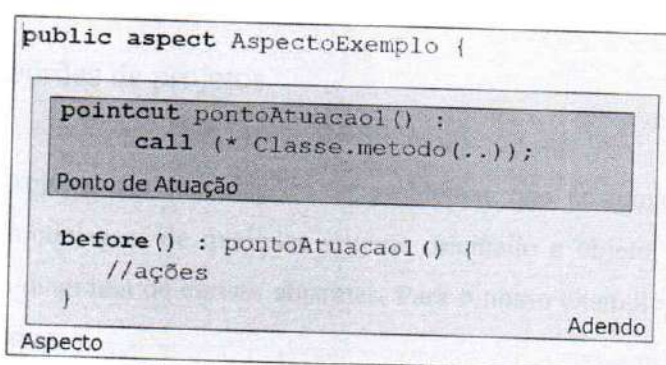


Figura 3 - Anatomia de um aspecto.

Fonte: Site aspects.org

3.3.3. Aspectos: com anotação X sem anotação

Em uma visão bem superficial, anotações são novos recursos que foram incorporados desde o JDK 5.0 e são definidos em detalhes na especificação JSR 175. Anotações permitem a você definir metadados de uma forma typesafe e aplicá-la a uma classe, método, construtor, campo ou parâmetro. A principal vantagem das anotações é que podemos dispensar o uso de arquivos xml para realizar configurações necessárias em nosso sistema.

Na programação orientada a aspectos, que é um paradigma relativamente novo, as annotations ainda não tem sintaxe para representar todas as funcionalidades já implementadas, sendo bem limitada quando se compara com o aspectos sem anotações.

A principal diferença entre um aspecto com e sem anotação, além das limitações, é que não se cria um objeto aspecto propriamente dito, e sim, uma classe anotada com `@Aspect`. Assim como o aspecto é uma classe, os pointcuts e advice são métodos normais anotados de forma específica para cada caso. Para os pointcuts utilizamos a anotação `@Pointcut` e para os advice já precisamos saber o seu tipo (before, after ou around) e anotar o método do modo apropriado, `@Before`, `@After` ou `@Around`.

```
@Aspect
public class Injetor2 {

    @Pointcut("@annotation(Injetor)")
    public void userUpdatingOperation() {

    }

    @Before("userUpdatingOperation()")
    public void doAccessCheck() {
        system.out.println("passei aqui 33");
    }
}
```

Figura 4 - Aspecto com anotação

Fonte: Resultado da pesquisa do autor.

3.3.4. Aspectos e padrões de projetos

Padrões de projetos são modelagens de problemas que costumam aparecer com certa frequência durante a modelagem de qualquer sistema orientado a objeto. Para representar esses padrões utilizamos um diagrama de classes abstratas. Para o nosso exemplo, vamos exemplificar o padrão de projeto observer:

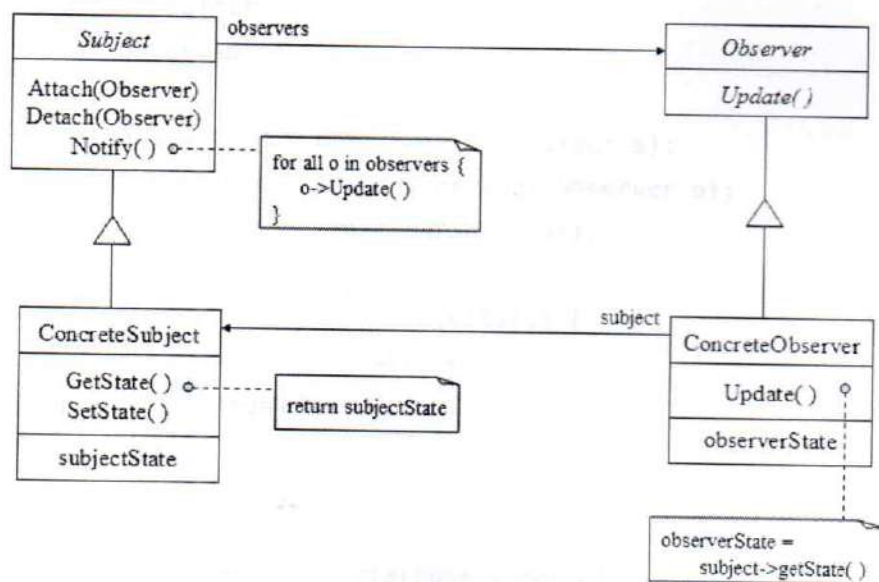


Figura 5 - Diagrama de classe do padrão Observer

Fonte: Site Aspects.org

Um problema com o uso de padrões é que os padrões estão embutidos no código, ou seja, podemos ver o padrão como um interesse entre cortante (transversal). Seria interessante se pudéssemos remover o padrão do código, na forma de um aspecto, e aplicá-lo às classes que desejássemos sem termos de modificar essas classes [NELSON, Torsten 2005].

Inicialmente, é preciso deixar claro que não é possível implementar o padrão Observer com o mesmíssimo comportamento do padrão definido acima, pois no padrão original cada observador "escolhe" qual o sujeito que vai observar e na POA não conseguimos fazer esse tipo de escolha. Sendo assim, vamos implementar uma versão do Observer em que todos os observadores vão se ligar ao mesmo sujeito [NELSON, Torsten 2005].

Nossa implementação usará *pontos de corte abstratos*. Esses são pontos de corte cuja definição será feita em uma subclasse do aspecto. O aspecto com o padrão tem toda a parte genérica, mas teremos de criar um aspecto concreto para especificar os observadores e sujeito específicos da nossa aplicação [NELSON, Torsten 2005].

Código 7 - Aspecto que simula o padrão de projeto observer

```

public abstract aspect PadraoObserver {

    protected interface Subject { }
    protected interface Observer {
        void update(Subject s);
    }
}
  
```

```

Subject sujeito;
List observadores;

abstract pointcut criarSujeito(Subject s);
abstract pointcut criarObservador(Observer o);
abstract pointcut modificarSujeito();

after(Subject s) : criarSujeito(s) {
    if (sujeito == null) {
        sujeito = s;
    }
}

after(Observer o) : criarObservador(o) {
    observadores.add(o);
}

after() : modificarSujeito() {
    Iterator i = observadores.iterator();
    while (i.hasNext()) {
        Observer o = (Observer) i.next();
        o.update();
    }
}
}

```

Agora precisamos criar um aspecto concreto e definir o sujeito que iremos observar. Vamos usar neste exemplo a classe `TocadorMusica` onde teremos duas formas de alterar o volume, uma pela barra *slider* que movemos com o mouse e um valor número para a porcentagem do volume, que podemos editar. Mudar um deve necessariamente mudar o outro. [NELSON, Torsten 2005]

Código 8 - Implementando o padrão observer com aspectos

```

public aspect ObserverConcreto extends PadraoObserver {
    pointcut criarSujeito(Subject s) : execution(MP3Player.new()) && this(s);
    pointcut criarObservador(Observer o) :
        (execution(Slider.new()) || execution(TextLevel.new())) && this(o);

    pointcut modificarSujeito() : execution(* MP3Player.setVolume(..));

    declare parents: MP3Player implements Subject;
}

```

```
declare parents: Slider implements Observer;
declare parents: TextLevel implements Observer;
```

```
public void Slider.update(Subject s) {
    MP3Player p = (MP3Player) s;
    position = p.getVolume();
}
```

```
public void TextLevel.update(Subject s) { ... }
```

O aspecto concreto define os pontos de corte correspondentes ao papel do sujeito (TocadorMusica) e observadores (TextLevel e Slider). Além disso, define que essas classes são subclasses das classes Subject e Observer, através da implementação de interfaces. Finalmente, implementa os métodos *update* que são responsáveis por lerem do sujeito os valores desejados e atualizarem cada observador [NELSON, Torsten 2005].

3.4. ASPECTJ

AspectJ é uma extensão da linguagem de programação Java que nos permite escrever código na linguagem aspecto. Por ser uma extensão do Java, o código escrito em aspectos será convertido em byte codes compatíveis com as especificações JVM. Essa compatibilidade só é possível porque os compiladores de AspectJ transformam as construções específicas de AspectJ em código Java puro, todas as construções de AspectJ que afetam classes Java modificarão o código destas classes utilizando construções de Java.

Em tempo de execução, alterações dinâmicas de um programa, o *AspectJ* executa os *Join Points*, os quais são selecionados pelos *Pointcuts* e através da adição de comportamentos (*Advice*) antes ou depois do *Join Point*, sempre seguindo esta ordem de seleção e execução. Também podemos alterar a estrutura estática de um programa adicionando membros (atributos, métodos ou construtores) a uma classe, podendo assim modificar a hierarquia do sistema.

3.4.1. Crosscutting elements

O AspectJ utiliza construções próprias para os blocos de aspecto: a implementação dos aspectos é construída em blocos que formam os módulos que expressam os interesses transversais. Os elementos básicos em AspectJ são: join points, pointcuts, advices, declarações inter-types e *aspects*. Já vimos os conceitos de *join point*, *pointcuts* e *advice* anteriormente, trabalhando agora com

AspectJ nos deparamos com alguns novos conceitos que precisamos entender como, *aspects*, *introduction* e *compile-time declaration*.

3.4.1.1. aspects

O aspect é a unidade central do AspectJ da mesma forma que a classe é a unidade central de Java. Nele estão implementadas as regras de construção de crosscutting dinâmico e estático. Pointcut , advice , introduction e declarations são combinados em um aspect . Como qualquer classe Java, os aspects podem conter atributos, métodos e classes internas.

Antes de começar a implementar o comportamento do crosscutting , precisa ser feita uma análise para identificar os join points e quais argumentos de comportamentos se desejam modificar, para que em seguida seja definido o novo comportamento. Para iniciar a implementação se escreve um aspect que serve como um modelo que contém uma implementação geral. Então, dentro do aspect , escrevem-se os pointcuts para capturar os join points desejados. Finalmente, se escreve os advice para cada pointcut e codifica-se a ação que se deseja executar quando o join point for chamado [HUGO, Marcel e GROTT Marcio Carlos, 2009].

3.4.1.2. introduction

Introduction é um crosscutting estático que introduz alterações nas classes, interfaces e em outros aspectos do sistema. Alterações estáticas em módulos não tem efeito direto no comportamento. Por exemplo, pode ser adicionado um método ou um atributo na classe em tempo de compilação.

3.4.1.3. compile-time declaration

O compile-time declaration é uma declaração de crosscutting estático que permite adicionar alerta e erro quando se detecta certos padrões de utilização de classes. Esse tipo de declaração crosscutting será muito utilizado neste trabalho na proposta da arquitetura onde serão definidas regras arquiteturais que, quando infringidas, marcarão o código com mensagens do tipo *warning* ou até mesmo *error*. Neste último caso, o aspecto não permitirá com que o projeto seja compilado até que o erro seja corrigido.

Um exemplo de compile-time declaration será quando houver uma instância (*new*) de uma classe DAO na camada de visão (*view*). Tal procedimento retornará uma mensagem de erro no

projeto e não será possível compilá-lo. Outro exemplo bem comum de acontecer é quando o programador deseja exibir no console da IDE o retorno de uma variável qualquer, fazendo o uso do comando `System.out.print`. Tal comando será considerado um *warning* pelo aspecto e deverá ser corrigido utilizando-se de uma biblioteca específica para este caso (log4j por exemplo), porém, este *warning* não irá restringir a aplicação a ponto de não permitir a sua compilação.

3.4.2. O compilador

O AspectJ oferece um conjunto de ferramentas para auxiliar na criação de aspectos, desde um compilador de aspectos, visualizadores de aspectos e plugins para integração com as mais populares IDE's de desenvolvimento.

O compilador é a unidade central da implementação da linguagem AspectJ. Ele é responsável pela combinação das classes Java com as unidades de aspectos produzindo os arquivos para serem utilizados em produção. O compilador de aspectos aceita arquivos de classes Java, unidades de aspectos ou uma mistura de ambos. Os arquivos resultantes da compilação contêm puro bytecodes Java e por isto pode ser executado em qualquer máquina virtual Java. Logo abaixo, segue um exemplo da combinação de uma classe Produto com dois interesses sistêmico, auditoria e controle de exceções:

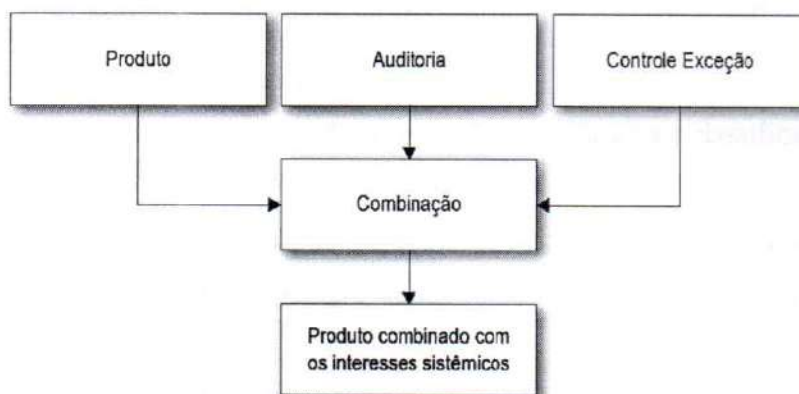


Figura 6 - Diagrama de compilação de interesses sistêmicos

Fonte: Resultado da pesquisa do autor.

O AspectJ oferece um conjunto de ferramentas para auxiliar na criação de aspectos, desde um compilador de aspectos, visualizadores de aspectos e plugins para integração com as mais populares IDE's de desenvolvimento.

3.4.2.1. weaver

Como acabamos de ver, aspectos são compostos pela mesclagem de código de aspecto com o código base. Este processo de mesclagem é conhecido como weaver. A imagem abaixo ilustra o processo de weaver dos métodos A e B com o Aspecto de distribuição:

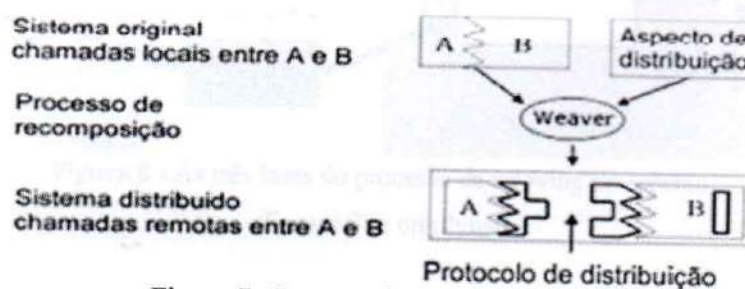


Figura 7 - Processo de weaving da POA

Fonte: Site javafree

O weaver age em um momento antes da compilação do código, procurando pelos pontos de junção que foram definidos pelos pontos de atuação e inserindo o comportamento que foi implementando nos adentos.

Ele pode atuar em tempo de compilação ou execução, permitindo a adição ou exclusão de aspectos enquanto a aplicação está em execução, dando um amplo campo de aplicação para o paradigma (FORTES, 2008).

A POA envolve três fases distintas:

- Decomposição: onde os interesses do sistema são identificados e classificados como comuns ou transversais;
- Implementação: os interesses são implementados separadamente em classes e aspectos. São definidas regras de corte, cada regra define onde deve ocorrer a interferência de algum interesse transversal;
- Recomposição: é nessa fase que ocorre o weaving. Os interesses gerados através de uma linguagem orientada a aspectos são mesclados como o código principal para formar o código executável final.

A imagem abaixo ilustra as três fases descritas acima:

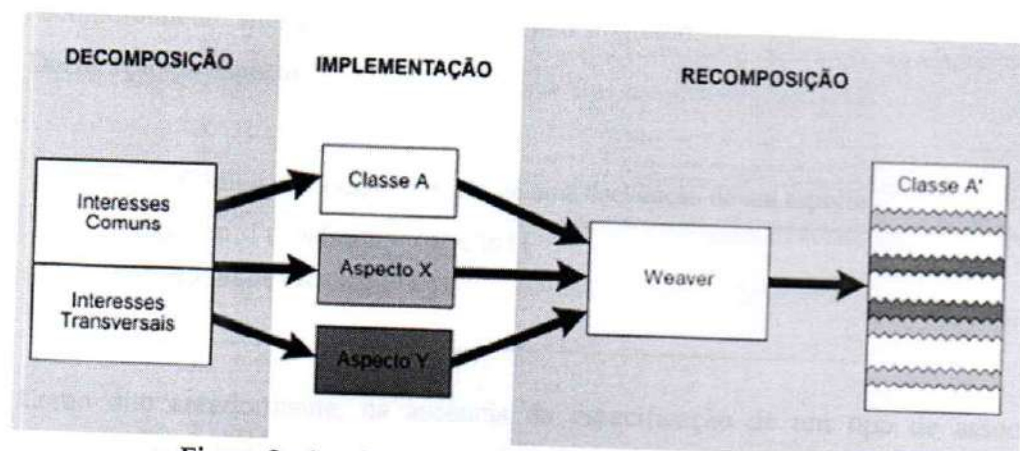


Figura 8 - As três fases do processo de weaving do aspecto.

Fonte: Site oreillynet

A tabela 6 descreve os tipos de weaving suportados pela POA.

Tipo	Descrição
Compile-time	O código-fonte da linguagem primária e de aspecto é mesclado antes de passar pelo compilador, onde o byte code é produzido.
Link-time	A mesclagem ocorre após o código da linguagem primária e de aspecto terem sido compilados em byte code.
Load-time	A mesclagem ocorre quando classes são carregadas pelo ClassLoader (objeto responsável por carregar dinamicamente as classes na máquina virtual).
Run-time	A máquina virtual é responsável pela detecção de pontos de junção e de carregar e executar os aspectos.

Tabela 6 - Tipos de weaving da POA

3.5. ASSOCIAÇÕES DE ASPECTOS

A associação de aspectos define o modo pela qual um determinado aspecto irá se relacionar com a aplicação. No AspectJ, o padrão para a associação é que na máquina virtual Java exista apenas uma instância do do aspecto, comportando-se de modo similar ao padrão *singleton* [WINCK, Diogo Vinícius].

Associações de aspectos se classificam em três categorias:

- por máquina virtual (padrão);
- por objeto;
- por fluxo de controle.

A associação por máquina virtual é a associação padrão de aspectos e por este motivo não necessita de nenhuma declaração específica para que o aspecto represente este comportamento. Neste tipo de associação, as entidades da máquina virtual compartilham apenas uma instância do

aspectos, comportamento similar ao padrão de projeto *singleton*.

Declaração do aspecto:

Código 9 - Código exemplo de uma declaração de um aspecto

```
aspect NomedoAspecto [tipoDeAssociação]{
    // corpo do aspecto
}
```

Como dito anteriormente, na ausência da especificação de um tipo de associação, o AspectJ a forma padrão. Entretanto, se o desenvolvedor deseja explicitar o tipo de associação, basta declarar o aspecto do seguinte modo:

Código 10 - Declaração de um aspecto explicitando o tipo de associação

```
aspect NomedoAspecto issingleton{
    // corpo do aspecto
}
```

3.5.1. Associações por objeto

A associação por objeto agrega uma nova instância de um aspecto com o objeto alvo ou em execução, obedecendo a regra definida no ponto de atuação. Há duas formas possíveis de associação por objeto: *perthis* e *pertarget* [WINCK, Diogo Vinícius].

A associação *perthis* agrega uma instância separada do aspecto, cujo objeto em execução coincida com o ponto de atuação definido entre parênteses:

Código 11 - Declaração de um aspecto com associação do tipo *perthis*

```
aspect NomedoAspecto perthis(pontoDeAtuação){
    // corpo do aspecto
}
```

Ao se utilizar a associação *pertarget*, uma instância separada do aspecto é agregada ao objeto alvo definido no ponto de atuação entre parênteses:

Código 12 - Declaração de um aspecto com associação do tipo *pertarget*

```
aspect NomedoAspecto pertarget(pontoDeAtuação){
    // corpo do aspecto
}
```

3.5.2. Associações por fluxo de controle

Associação por fluxo de controle é um conjunto de operações que pode ser abstraído como um objeto que encapsula um processo em torno do ponto de atuação definido. Existem duas formas deste tipo de associação: `percfow` e `percfowbelow`.

Utilizando a associação `percfow`, uma instância separada é agregada ao fluxo de controle no ponto de junção que coincida com o ponto de atuação definido entre parênteses:

Código 13 - Declaração de um aspecto com associação do tipo `percfow`

```
aspect NomedoAspecto percfow(pontoDeAtuação){  
    // corpo do aspecto  
}
```

Utilizando a associação `percfowbelow` uma instância separada do aspecto é agregada ao fluxo de controle abaixo do ponto de junção que coincida com o ponto de atuação definido entre parênteses:

Código 14 - Declaração de um aspecto com associação do tipo `percfowbelow`

```
aspect NomedoAspecto percfowbelow(pontoDeAtuação){  
    // corpo do aspecto  
}
```

3.6. ONDE USAR ASPECTOS

Para não torna muito complexo o entendimento da aplicação, podemos definir em que situações utilizaremos os recursos que a POA nos oferece.

3.6.1. Controle de exceções

Uma exceção é algo que foge do seu fluxo normal esperado em um sistema. Quando uma exceção é lançada, é esperado que o método em execução capture tal exceção realizando os tratamentos necessários. Se o método em execução não capturar a exceção, esta será passada para o método chamador e assim por diante até que seja tratada ou que saia do programa principal.

Quando um método gera uma exceção, diz-se que ele lança (*throws*) a exceção. Em Java, usa-se o comando *throw*. A assinatura do método deve declarar todas as exceções jogadas por ele.

Código 15 - Método lançando um exceção

```
void lerDisco( ) throws IOException {
    if (problemaDetectadoDisco) {
        throw (new IOException() );
    }
}
```

Para capturar a exceção lançada utilizamos blocos *try..catch*. Para capturar a exceção, o método tem que ser chamado dentro do bloco *try*. Se a exceção for lançada, o controle passa para o bloco *catch*. Segue um exemplo de captura de exceção:

Código 16 - Método com tratamento de exceção

```
void strToNumber(String s) {
    try {
        int num;
        num = Integer.parseLong(s);
    } catch (NumberFormatException e) {
        System.out.println("Erro: o string não é um número válido");
        e.printStackTrace();
    }
}
```

Se a exceção não é tratada do modo em que foi descrito acima, devemos lançá-la (*throws*) para a camada superior da aplicação:

Código 17 - Método lançando exceção para camadas superiores

```
void strToNumber(String s) throws NumberFormatException {
    int num;
    num = Integer.parseLong(s);
}
```

Como podemos perceber, em todo o sistema, quando fizermos uso de uma método que lança uma exceção, teremos que tratar as exceções de alguma forma, ou lançando-a para uma camada superior ou tratando com os blocos *try..catch*. Tal processo de tratamento gera duplicidade de código pois o tratamento de exceção é um interesse entre cortante da aplicação, sendo assim podemos criar um aspecto que tratará todas as exceções do sistema:

Código 18 - Aspecto que captura e trata exceções

```
public aspect Excecao {  
    // Captura a exceção do método, exceto do pacote do próprio aspecto  
    pointcut tratamentoExcecao(): handler(Exception+)  
    && !within(aspectos.*);  
  
    // Comportamento de tratamento da exceção  
    before() : tratamentoExcecao(){  
        Logger l=Logger.getLogger("Console-Error");  
        l.log(Level.SEVERE, "Log da Exceção sendo tratada");  
    }  
}
```

O aspecto acima define qual pointcut o aspecto vai usar (tratamentoExcecao()). Este pointcut mapeia o ponto de interesse (joinpoint) de tratamentos de Exception (handler) e de todas as suas descendentes (+). O comando !within na expressão certifica que o pacote dos aspectos não será alvo do próprio aspecto, impedindo ciclos.

3.6.2. Restrições arquiteturais

Fazendo uso de crosscutting static, podemos adicionar restrições arquiteturais em nossa aplicação. Ao desenvolver uma arquitetura, além de provermos os serviços essenciais e básicos para uma aplicação, também podemos realizar restrições de como a aplicação fará uso da arquitetura e de como a aplicação será implementada.

Em uma arquitetura de software podemos escrever nossos aspectos para que fique “observando” como o programador está desenvolvendo o sistema, se alguma regra arquitetural que foi definida no aspecto for quebrada, imediatamente, em tempo de compilação, o programador será alertado e o problema deverá ser corrigido.

Imagine o programador implementando um *managed bean*, nas boas práticas de programação é dito que a camada de visão não deve chamar a camada de persistência diretamente e o programador, por descuido ou até mesmo por falta de capricho, acaba chamando o DAO diretamente. Em uma arquitetura comum esse comportamento no ato da programação acaba passando e a qualidade da programação fica prejudicada.

Com o *compile-time declaration* podemos expressar *pointcuts* que observam um pacote específico, pacote DAO, por exemplo. Também podemos observar o pacote da visão e fazer uma verificação de instâncias (*new*) em cada uma das classes desses pacotes. Sendo assim podemos criar regras onde nas classes do pacote da camada de visão não poderá receber nenhuma chamada de classes do pacote de persistência. Caso isso ocorrer, declaramos erro e o projeto não poderá ser

3.6.3. Transação e persistência de dados

Quando falamos de persistência de dados, um dos conceitos que nos trás muitos benefícios é o conceito de transação em banco de dados que nada mais é do que a capacidade de restaurar o estado anterior dos dados, caso ocorra alguma falha dentro do contexto transacional. Na realidade, isso é possível porque os dados não são persistidos, de fato, até que a transação seja confirmada através do comando *commit*, ou seja, enquanto a transação não for “commitada” esses dados ficam numa área de armazenamento temporária que é visível apenas à sessão corrente, mas não aos demais usuários do BD.

Abaixo vamos exemplificar como implementamos as funcionalidade de ações transacionais em um BD:

Código 19 - Exemplo de métodos transacionais

```
try {  
    transaction.beginTransaction();  
    metodo1();  
    metodo2();  
    ...  
    metodoN();  
    transaction.commit();  
} catch (Exception e) {  
    transaction.rollback();  
}
```

Neste trecho de código acima podemos observar que antes dos métodos que interagem com o meu BD, iniciamos uma transação com o comando *transaction.beginTransaction()*. Se tudo ocorrer dentro do previsto, todos os métodos serão executados e o comando *transaction.commit()* será chamado, efetivando as mudanças no banco de dados. Caso alguma exceção seja levantada, o fluxo de execução será deslocado para o bloco *catch* executando o comando *transaction.rollback()* e restaurando os valores originais dos objetos.

Diante desses benefícios que a transação nos oferece, é interessante que boa parte de nossa aplicação seja transacionada o que nos levaria a repetir código em todas as classes que queremos utilizar tal recurso.

Para evitar essa duplicidade de código podemos utilizar os recursos da POA, como exemplificado abaixo:

Código 20 - Aspecto que transaciona operações

```
// Procurando por métodos anotados com @Transacional
pointcut transacao(): (@Transacional * *.*(..));

// Antes do método anotado, inicie a transação
before() : transacao(){
    JPAUtil.beginTransaction();
}

// Se não ocorrer nenhum erro, commit as informações
after()returning : transacao() {
    JPAUtil.commitTransaction();
    JPAUtil.closeEntityManager();
}

// Se alguma exceção for lançada, execute o rollback
after() throwing : transacao() {
    JPAUtil.rollbackTransaction();
}
```

O aspecto acima está procurando em toda a minha aplicação métodos que estejam anotados com a anotação `@Transacional`, e iniciando antes (*before()*) a transação (*beginTransaction()*). Caso o método execute tudo sem nenhum erro (*returning*), depois do método (*after()*) chamamos o commit para efetivar minha transação. Caso contrário, o método levanta uma exceção (*throwing*) e chamamos o rollback, restaurando os valores dos objetos alterados.

3.6.4. Auditoria

Uma auditoria é o processo de registrar as ações feitas por um usuário afim de provar as ações tomadas por ele. Dessa forma é possível provar que o sistema não estava errado. Por exemplo, os dados sumiram porque alguém os excluiu.

A auditoria também se enquadra em um interesse de suporte ao sistema, sendo um interesse que transpassa por toda a aplicação e, conseqüentemente, terá código espalhada se programada de maneiras tradicionais.

Segue abaixo um exemplo da classe DAO cliente e suas funcionalidades que serão auditadas:

Código 21 - Classe ClienteDAO

```

public class ClienteDAO implements EntityDAO<Cliente>{
    public void atualizar(Cliente c){
        entityManager.merge(c);
        entityManager.flush();
    }

    public void inserir(Cliente c){
        entityManager.persist(c);
        entityManager.flush();
    }

    public void remover(Cliente c){
        entityManager.remove(c);
        entityManager.flush();
    }
}

```

Utilizando-se da programação tradicional, o modo que auditaríamos as ações desta classe seria de seguinte maneira:

Código 22 - Classe ClienteDAO com a implementação de auditoria

```

public class ClienteDAO implements EntityDAO<Cliente>{
    // Código que precisou ser inserido na classe
    private Auditoria aud = Auditoria.getLoadClass(ClienteDAO.class);

    public void atualizar(Cliente c){
        entityManager.merge(c);
        entityManager.flush();
        aud.gravaLog(MSG_ATUALIZAR_ENTIDADE_SUCESSO);
    }

    public void inserir(Cliente c){
        entityManager.persist(c);
        entityManager.flush();
        aud.gravaLog(MSG_CADASTRAR_ENTIDADE_SUCESSO);
    }

    public void remover(Cliente c){
        entityManager.remove(c);
        entityManager.flush();
        aud.gravaLog(MSG_REMOVER_ENTIDADE_SUCESSO);
    }
}

```

```
}
}
```

Como podemos observar, programando tradicionalmente este interesse teremos em todas as nossas entidades que implementarem a interface EntityDAO código repetido da implementação da funcionalidade de auditoria. Neste exemplo vimos apenas como uma classe é afetada pela adição desta funcionalidade, na figura 9 temos um exemplo de como ficaria em todo o sistema.

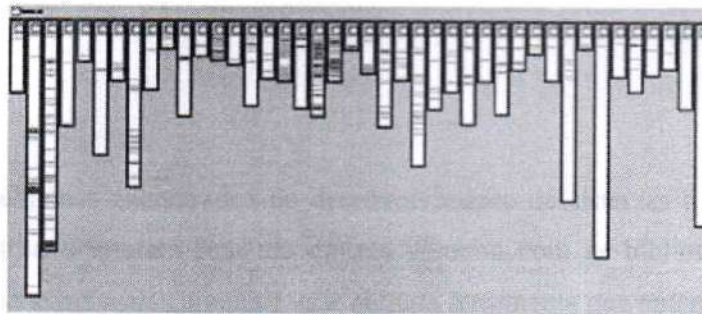


Figura 9 - Interesses transversais espalhados pelo sistema.

Fonte: Site aspects.org

Cada barra branca representa uma classe do sistema em análise, o tamanho da barra representa a quantidade de linhas programadas e os riscos destacados nas barras representam a funcionalidade de auditoria que, como podemos observar, está completamente espalhada pelo sistema, tornando muito complicada a manutenção desta funcionalidade.

Sabendo que as entidades que implementam a interface EntityDAO terão a implementação dos métodos de persistência, podemos realizar a auditoria desses métodos via aspecto da seguinte maneira:

Código 23 - Aspecto que implementa auditoria

```
public aspect AuditoriaEntity {

    private static final Logger logger = Logger.getLogger(AuditoriaEntity.class);

    pointcut publicMethods() : execution(public * ..dao..*(..));
    pointcut loggableCalls() : publicMethods();

    private Logger getLogger(org.aspectj.lang.JoinPoint joinPoint) {
        try {
            Class declaringType = joinPoint.getSignature().getDeclaringType();
            Field loggerField = declaringType.getField("logger");
            loggerField.setAccessible(true);

            return (Logger)loggerField.get(null);
        } catch (NoSuchFieldException e) {
            return logger;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

```

    }
}

before() : loggableCalls(){
    getLogger(thisJoinPoint).debug("Entrando.." +
    thisJoinPoint.getSignature().toString());
}

after() : loggableCalls(){
    getLogger(thisJoinPoint).debug("Saindo.." +
    thisJoinPoint.getSignature().toString());
}
}

```

3.7. MAVEN

Um dos problemas encontrados no desenvolvimento de sistemas é como fazer com que toda a equipe construa o artefato final da mesma maneira com as bibliotecas e configurações corretas. Uma das maneiras mais comuns é usar alguma ferramenta que entenda alguma linguagem de script e copie as dependências e configurações para os lugares corretos, no caso da linguagem Java podemos utilizar o Ant. Com o Ant é possível escrever scripts de 'build' para copiar bibliotecas de lugares específicos, arquivos de configurações e qualquer outro tipo de recurso para o artefato da aplicação. Com isso garantimos que qualquer desenvolvedor consiga gerar o artefato da aplicação localmente, sabendo que este artefato gerado é o mesmo que o resto da equipe irá usar e que possivelmente será usado em produção [MOREIRA, Gustavo 2010].

Nesta abordagem encontramos alguns problemas. A solução mais comum é guardá-las no repositório junto com o código fonte, assim todos os desenvolvedores compartilharão as mesmas dependências e os caminhos das pastas são mantidos. Esse tipo de organização é muito adotada e funciona muito bem para projetos pequenos e com equipes bem pequenas. Quando começamos a prolongar um pouco a vida do projeto percebemos que o *overhead* de manutenção do mesmo começa a interferir no andamento do desenvolvimento. Isso porque o número de dependências começa a aumentar e com o tempo vão saindo versões novas de algumas dependências antigas (podemos entender dependências como sendo bibliotecas). Se todas as dependências estão na mesma pasta, fica muito difícil fazer o controle de versão e de transitividade das mesmas [MOREIRA, Gustavo 2010].

Agora pense em um caso mais complicado: decidimos criar um novo módulo para o projeto. Nesse caso, todas as dependências que este novo módulo necessita e estão no módulo já existente, devem ser duplicadas no repositório. Deve-se fazer isso pois já que estamos falando de módulo, devemos ser capazes de construir o tal módulo independente da existência dos outros. Ou seja, duplicamos também o esforço para manter a organização das dependências.

Essa é uma das motivações principais pela qual a apache criou o Maven. O Maven é uma ferramenta de integração de projetos. É responsável por gerenciar dependências, controlar versão de

artefatos, gerar relatórios de produtividade, garantir execução de testes, manter nível de qualidade do código dentre outras. Muitas pessoas confundem o maven como sendo uma alternativa para o Ant, ou para o Ivy o que não é verdade. O Maven inclusive disponibiliza a funcionalidade de rodar arquivos do Ant durante o build [MOREIRA, Gustavo 2010].

O que torna o Maven muito poderoso é a facilidade que ele fornece para se trabalhar com vários módulos de um mesmo sistema e sua extensibilidade para novas funcionalidades com o uso de 'plugins'. Existem plugins de geração de código, de integração com plataformas de teste e inclusive suporte a IDEs como Eclipse e NetBeans. Isso torna o projeto muito mais flexível dentro da equipe, pois cada desenvolvedor pode escolher a IDE com que vai trabalhar sem se preocupar em atrapalhar o resto da equipe.

3.7.1. Fase do build

Uma fase nada mais é do que um estágio onde são executadas algumas regras sobre o projeto e se obtém algum resultado no final. Por exemplo, a fase de testes roda os testes da aplicação e obtém um 'OK' ou um 'FAIL', no segundo caso o build é interrompido. Tais fases são executadas dentro do 'lifecycle' do build. O 'lifecycle' é composto de 'goals' e são estes:

- **validate**: valida que os 'poms' dos projetos envolvidos estão corretamente escritos e que todas as informações necessárias para o build estão presentes;
- **compile**: compila todos os códigos do projeto, inclusive os códigos das classes de teste;
- **test**: roda os testes que estão em 'src/test/java', e certifica-se que todos estão passando, caso contrário o build é interrompido;
- **package**: usa o código compilado e testado que está em 'src/main/java' e cria um arquivo reusável, por exemplo jar;
- **integration-test**: nesta fase serão rodados os testes que necessitam do jar do projeto 'deploiado' para serem rodados;
- **verify**: roda as verificações necessárias para se certificar que os pacotes gerados estão corretos e passam nos critérios de qualidade;
- **install**: copia o arquivo gerado para o repositório local para que esteja disponível localmente para outros projetos;
- **deploy**: copia o arquivo gerado para um repositório na rede ou remoto, para que esteja disponível para outros desenvolvedores.

3.7.2. Project object model (pom)

Project Object Model se trata do ponto de partida para o maven executar seu lifecycle. Nada mais é do que um arquivo XML que descreve propriedades e características do projeto, como por exemplo a versão do compilador que será usada:

```

<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>br.com.flopes</groupId>
  <artifactId>teste-maven</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>teste-maven</name>
  <url>http://maven.apache.org</url>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <compilerVersion>1.6</compilerVersion>
          <source>1.6</source>
          <target>1.6</target>
          <encoding>UTF-8</encoding>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>

```

Figura 10 - Estrutura simples de um arquivo POM

Fonte: Resultado da pesquisa do autor

Repare que todas as propriedades que foram escolhidas no momento em que geramos o projeto estão definidas aqui, por isso se for preciso editar alguma delas, podemos fazer direto aqui no XML.

3.7.3. Dependências

Uma das funcionalidades mais poderosas do maven é a resolução de dependências. Não é uma funcionalidade exclusiva, mas a maneira com que é feita resolve muito bem a maioria dos problemas. Existem algumas outras ferramentas com o mesmo intuito como, por exemplo, o Ivy da própria Apache, mas vimos que o maven é muito mais do que apenas uma ferramenta de resolução de dependências.

A definição das dependências do seu projeto é feita inteiramente no pom. Para isso usamos a tag <dependencies/> e dentro definimos cada dependência individualmente. Por exemplo, para definirmos a dependência do JUnit 4.8.1 basta adicionar o seguinte trecho no lugar do que já existe:

```

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.8.1</version>
  <scope>test</scope>
</dependency>

```

Figura 11 - Declaração das dependências no arquivo POM

Fonte: Resultado da pesquisa do autor

A tag `scope` explicita o escopo em que o artefato definido será usado, nesse caso estamos dizendo para o maven adicionar a dependência do JUnit apenas no momento em que os testes forem executados, garantindo assim que a dependência não seja adicionada no artefato que será gerado.

Alguns outros escopos:

- **compile:** escopo padrão do maven para o momento em que o código é compilado e vai junto com o artefato;
- **provided:** adicionado no momento da compilação, mas não vai junto com o artefato. Dessa forma você está dizendo que a dependência virá de maneira transitiva de alguma outra dependência;
- **runtime:** não inclui no artefato, pois estará disponível em tempo de execução;
- **test:** inclui apenas no escopo de testes;
- **system:** não inclui no artefato, pois estará disponível no ambiente;
- **import:** incluirá todas as dependências do '**dependencyManagement**' que está definido no pom 'parent'.

3.7.4. Dependency Management

O dependency management é responsável por controlar as versões das dependências definidas no projeto. Por exemplo podemos definir a dependência do JUnit no dependency management, e apenas referenciar a dependência na tag `dependencies` do projeto:

```

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.1</version>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
  </dependency>
</dependencies>

```

Figura 12 - Declaração de um dependency-Management no arquivo POM

Fonte: Resultado da pesquisa do autor

Uma boa prática é definir todas as versões das dependências em um dependency management do pom parent do seu projeto. Dessa forma todo projeto novo que surgir, basta herdar desse parent que as versões já estarão compartilhadas e continuarão centralizadas em um único projeto. Mesmo com a dependência definida no management, os projetos ainda podem definir suas próprias versões de artefatos e o maven dará prioridade para os mais próximos do projeto.

3.7.5. Properties

Outro recurso que está disponível no maven e podemos utilizar são as properties. Properties nada mais são do que declaração de variáveis, onde pode-se criar essas variáveis com qualquer nome que se desejar e utilizá-la em qualquer parte do pom.

```
<properties>
  <spring.version>3.0.6.RELEASE</spring.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>${spring.version}</version>
    <type>jar</type>
  </dependency>
</dependencies>
```

Figura 13 - Utilizando properties no arquivo POM

Fonte: Resultado da pesquisa do autor

Como descrição na imagem acima, dentro da tag properties declaramos uma variável spring.version (lembrando que pode ser qualquer nome que desejar) e definimos um valor. Este valor pode ser qualquer string, no nosso caso utilizamos para definir a versão do spring que iremos utilizar. Logo abaixo da properties, nas dependências do projeto, declaramos a dependência do projeto normalmente, havendo alteração apenas na tag version onde iremos chamar a nossa propriedade spring.version com a instrução \${spring.version}.

3.8. ARQUITETURA DE SOFTWARE

O problema de se construir um sistema é que, quanto mais ele cresce, mais ganha complexidade. Diante desta problemática que atinge boa parte do desenvolvimento, construir um sistema vai além da escolha de algoritmos e estrutura de dados, precisando tomar decisões sobre a estrutura que formarão o sistema, a estrutura global de controle ou ainda protocolos de

comunicação, sincronização e acesso a dados. Todos esses elementos impactarão no comportamento, escalabilidade e desempenho do sistema.

Desenvolver uma arquitetura com este nível de maturidade requer tempo. Neste trabalho de conclusão de curso iremos apresentar uma arquitetura em seu estágio inicial de desenvolvimento, onde utilizaremos a programação orientada a aspectos para resolver percalços do que por ventura possam aparecer no desenvolvimento do software.

3.8.1. O que é uma arquitetura de software

Desde de 1970 vários autores se propuseram a definir o termo arquitetura de software e diante de pensamentos diferentes tentando expressar com palavras um mesmo significado, criou-se o padrão ISO/IEEE 1471-2000 para ajudar no consenso entre autores, estudantes e profissionais sobre o que é e para que serve arquitetura de software.

“Arquitetura é a organização fundamental de um sistema incorporada em seus componentes, seus relacionamentos com o ambiente, e os princípios que conduzem seu design e evolução.” [IEEE, and ISO/IEC].

3.8.2. Maturidade arquitetural

Ao desenvolver este projeto foi constatado que ao longo do desenvolvimento de software utilizando, a arquitetura amadurece ganhando classes cada vez mais abstratas e genéricas, aumentando seu escopo de atuação e abrangendo mais facilmente qualquer tipo de software que venha a ser implementado. Quando mais você utiliza a arquitetura, o refinamento aumenta e sua maturidade se eleva.

Segundo Paulo Alvim descreveu em seu livro Tirando o Máximo do JAVA EE 5 Open-Source, uma arquitetura de software pode ser classificada quanto a sua maturidade.

3.8.2.1. arquitetura de software “invisível”

Segundo Paulo Alvim, em empresa que desenvolvem softwares dificilmente não existirá uma arquitetura que seja realmente visível para os implementadores. Neste nível de maturidade o que se chama de arquitetura são alguns documentos, traçando diretrizes conceituais para o uso do MVC, utilizando um ou outro framework e fazendo uso de alguns designs patterns.

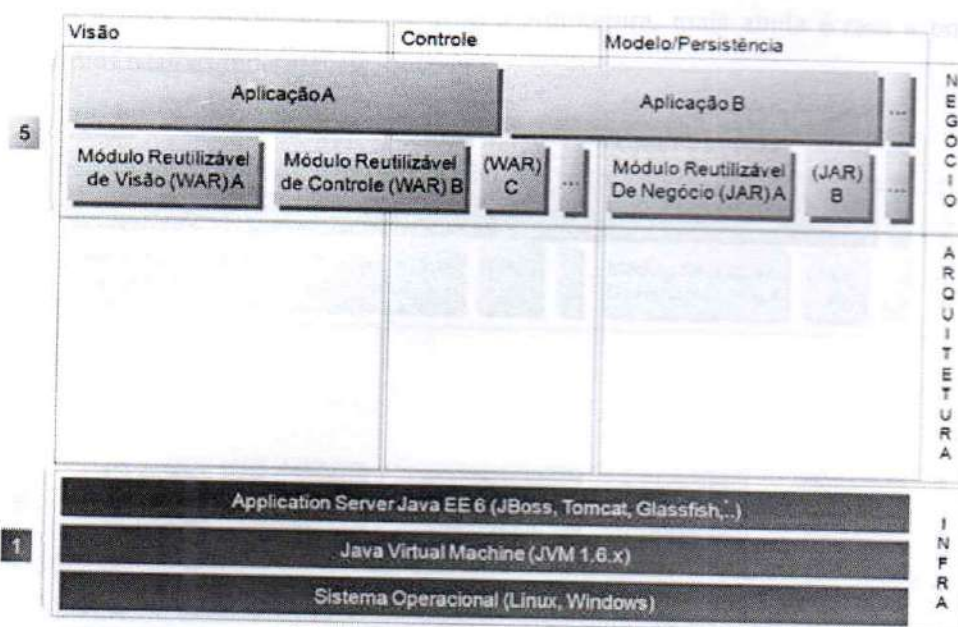


Figura 14 - Arquitetura invisível

Fonte: Imagem retirada do livro de Paulo Alvim

Este cenário é primitivo, insuficiente para se perceber qualquer ganho considerável. Neste estágio classificamos a arquitetura como “Arquitetura de software invisível”.

Como podemos perceber na imagem acima, existe uma lacuna vazia entre a infra [1] e as aplicações de negócio [5]. Por este motivo, um enorme potencial para erros está à espreita dos desenvolvedores, que ficam às voltas com programações repetitivas de código, que deveriam estar fatoradas em nível arquitetural.

3.8.2.2. arquitetura de software “anêmica”

Segundo Paulo Alvim, este nível de maturidade é um pouco mais adiantado, onde a arquitetura já se apresenta como software podendo ser reutilizada. Essa arquitetura geralmente é composta por frameworks de base open-source integrados que foram definidos após algum tempo de prospecção, seleção e integração “leve”.

Embora este nível seja mais elevado, ainda temo problemas com relação ao *Core* do negócio que os programadores terão que lidar:

- Código MVC “burros” (de => para, fachadas artificiais);
- Gerenciamento de transações manuais;
- Código Java para inclusões, alterações e exclusões (CRUD);
- Elaboração de formulários “do zero”;
- Gestão de layouts manuais.

Há um avanço perceptível com relação a arquitetura, mais ainda é rasa a ponto de não eliminar problemas básicos indesejáveis.

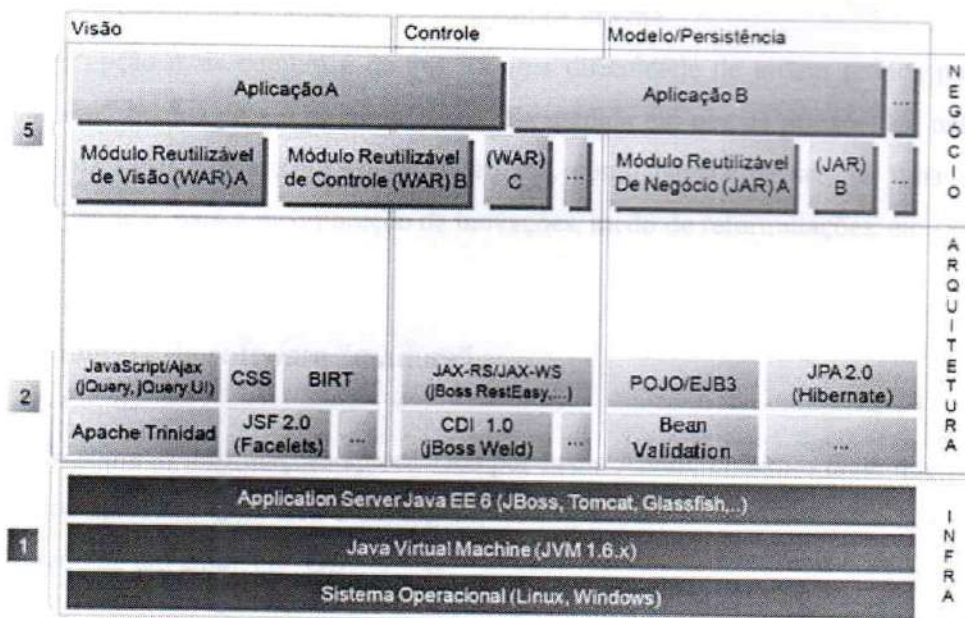


Figura 15 - Arquitetura anêmica

Fonte: Imagem retirada do livro de Paulo Alvim

3.8.2.3. arquitetura de software “intermediária”

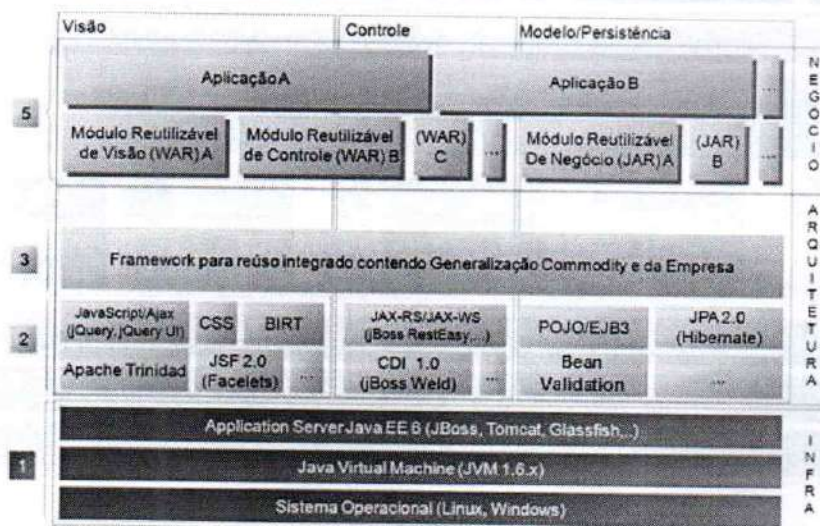


Figura 16 - Arquitetura intermediária

Fonte: Imagem retirada do livro de Paulo Alvim

Paulo Alvim afirma que neste nível já encontramos evidência clara de uma primeira “arquitetura de software corporativa”, integrando um bom número de insumos (frameworks e utilitários) atingindo diversas camadas do MVC.

Paula ainda afirma que para tal nível de maturidade é necessário uma comprovação de uso

satisfatório dessa arquitetura em um período mínimo de dois anos após a primeira versão lançada. A partir deste ponto o modelo começa a apresentar suas limitações, especialmente se a implementação arquitetural for desenvolvida totalmente por uma equipe interna.

A percepção mais comum é de que há uma dificuldade de ordem prática para que esta solução “familiar” evolua para o próximo passo. Na medida em que as pressões de negócio sobre aplicações implantadas prevalecem como prioridade, como haverá de ser, há uma tendência à estagnação ou lentidão para a incorporação de inovações, medo de reformulações, etc.

3.8.2.4. arquitetura de software “madura”

Ainda sobre as informações apresentada por Paulo Alvim, podemos dizer que este nível de maturidade nos apresenta uma rica e abrangente configuração, o suficiente para nos oferecer um retorno esperado de um ambiente OO.

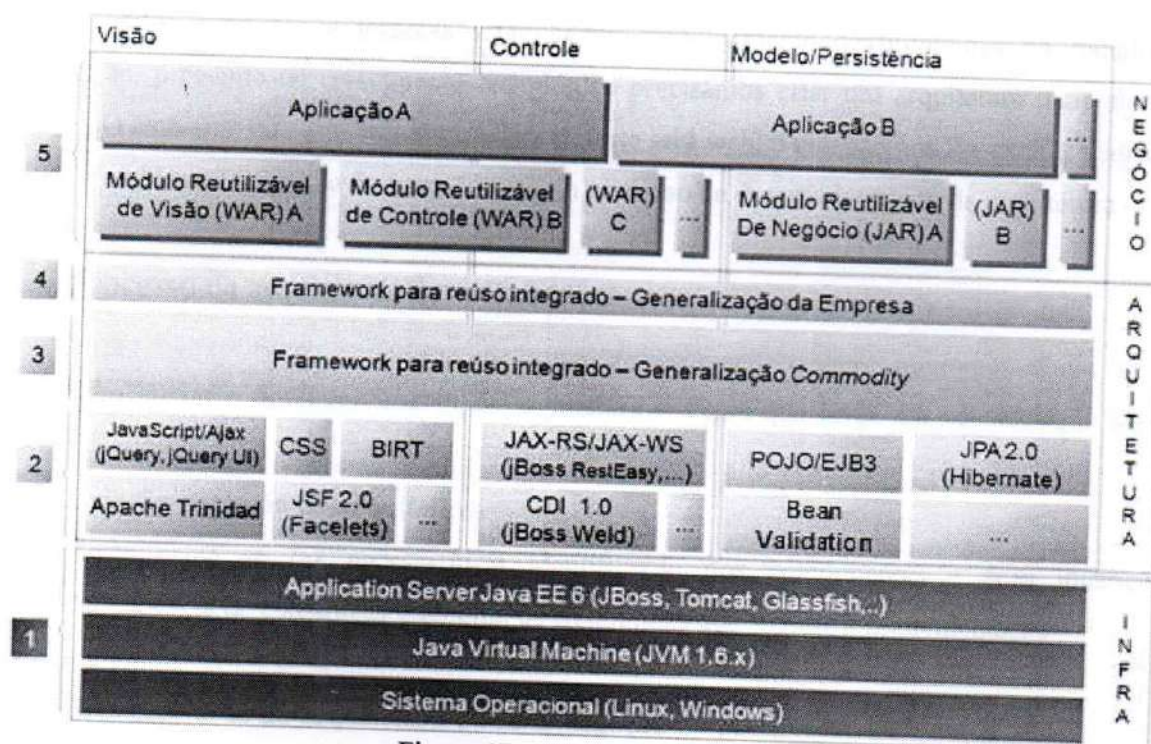


Figura 17 - Arquitetura madura

Fonte: Imagem retirada do livro de Paulo Alvim

Neste estágio, as preocupações de nível arquitetural que não sejam absolutamente dependentes do contexto específico da empresa são totalmente desacopladas de preocupações de natureza comum.

Este é o primeiro modelo que oferece um importante raciocínio de negócios: a potencial terceirização das camadas arquiteturais “2” e “3”, horizontais, de forma a possibilitar a

concentração da organização nas camadas "4" e "5". De fato, somente as duas últimas têm relevância para o ambiente específico de TI e verticais de negócio.

Com esta possibilidade, aumenta-se não somente o foco no negócio. Diminui-se o custo e herda-se qualidade e maturidade, abreviando-se longos anos e um grande número de problemas.

4. ESTUDO DE CASO

4.1. PROPOSTA DE ARQUITETURA

A proposta central deste trabalho é utilizar os benefícios que a programação orientada a aspectos nos fornece na construção de uma arquitetura de software, analisando cada ponto onde o aspecto ajudará não somente na arquitetura mas também no desenvolvimento do software que irá utilizá-la.

Novos conceitos e técnicas estão sendo continuamente desenvolvidas no mundo da computação, prevendo tal crescimento tecnológico precisamos criar um arquitetura mais abstrata possível. O ambiente que esta arquitetura dará suporte será web, o que não nos faz excluir a ideia de suporte a ambientes móveis e desktop. Vamos ao processo de desenvolvimento da arquitetura.

4.1.1. Processo da arquitetura de software

Ao iniciar o desenvolvimento da arquitetura teremos que levar em consideração alguns tópicos:

- **Criação e seleção da arquitetura**
Nesta etapa se faz necessário a identificação dos componentes e suas interações, dependência de construção e escolha de tecnologias que suporte a implementação.
- **Representação da arquitetura e sua divulgação**
Os desenvolvedores que utilizarem a arquitetura deverão entender como ela funciona para realizar os trabalhos que lhes foram atribuídos
- **Implementação do sistema baseado na arquitetura**
Os desenvolvedores devem se restringir às estruturas e protocolos definidos na arquitetura. Este ponto será atacado fortemente neste trabalho utilizando aspectos para realizar este tipos de restrição

- **Análise ou avaliação da arquitetura**

Deve ser verificada a adequação da arquitetura, registrando impactos, risco e dificuldades, uma vez que essas informações irão contribuir para a evolução da arquitetura em novas versões.

4.1.2. Estrutura arquitetural

Nesta primeira versão da arquitetura, foram identificados alguns pacotes que poderiam conter classes genéricas que dariam suporte à aplicação. Pacotes com os aspectos, de negócio, comparators, interfaces, superclass entre outros que serão citados e explicados a seguir:

- **Pacotes dos aspectos (br.com.flopes.arquitetura.aspects)**

Neste pacote ficarão os aspectos que darão suporte tanto para a arquitetura quanto ao sistema que será escrito utilizando a arquitetura. Para os aspectos agir de forma eficiente, o sistema terá que seguir algumas regras de criação de pacotes, como nomenclatura padrão (para a camada de persistência, deverá utilizar um pacote com o nome “dao”, por exemplo) para que o aspectos consiga através das regras escritas nos pontos de atuação selecionar os pontos de junções pertinentes.

- **Pacote de negócio (br.com.flopes.arquitetura.business)**

A princípio neste pacote foram identificadas três classes genéricas que o compoem. A classe BusinessLogic, uma classe genérica que armazena os critérios de seleção e solicitações, também foi criada a BusinessException para levantar as exceções do tipo BusinessLogic e a classe BusinessUtil onde se tem alguns métodos utilitários. Segue o diagrama de classes deste pacote:

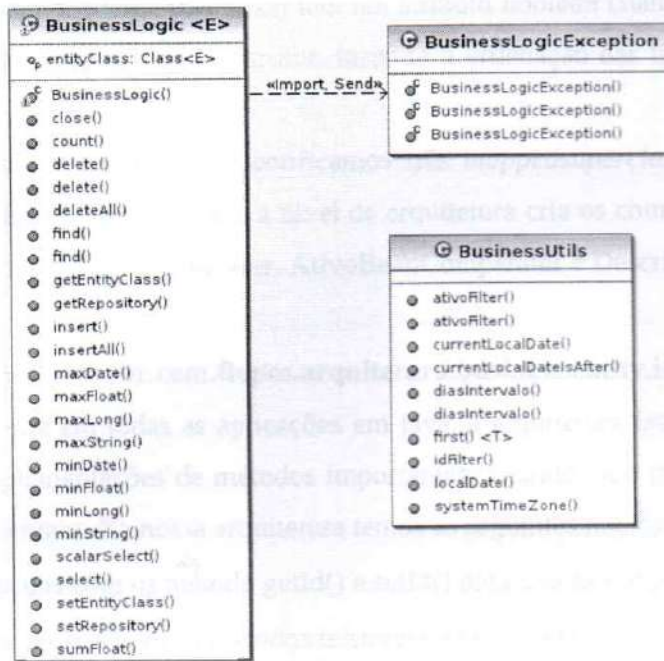


Figura 18 - Diagrama das classes de negócio

Fonte: Resultado da pesquisa do autor

- Pacote dos mappedSuperClass (br.com.flopes.arquitetura.business.entity.mappedsuperclass)

Algumas vezes temos entidades que possuem propriedades em comum e para não repeti-los utilizamos o recurso de superclasse, onde os atributos em comum são transferidos para a superclasse para que as entidades filhas possam herdá-los evitando a duplicidade de código. Olhando para as aplicações que irão utilizar a arquitetura, foi possível extrair três atributos básicos que as entidades provavelmente terão em sua implementação: id, ativo e descrição. Diante deste cenário, foram criadas três superclasses para suprir nossas necessidades. A primeira foi a `IdMapped`, uma classe que implementa a interface `IdBean` (falaremos dela a mais adiante) e por consequência, implementamos os métodos getters e setters do atributo "id". Nesta superclasse também foram feitos override de três métodos importantes para as entidades de um sistema: `hashCode`, `equals` e `toString`. A segunda superclasse implementada foi a `IdAtivoMapped` que estende a superclasse `IdMapped` e tem apenas o atributo "ativo" e seus métodos `get` e `set`. A terceira superclasse foi a `IdAtivoDescricaoMapped` que, assim como na `IdAtivoMapped`, tem apenas um atributo, chamado "descricao" e seus métodos `get` e `set`.

- Pacote dos comparators (br.com.flopes.arquitetura.business.entity.comparator)

Neste pacote foi criado um comparador abstrato, que implementa a interface `Comparator` sem assim obrigado a implementar o método `compare` para ordenarmos listas de objetos. O

AbstratoComparator (nome da classe) tem um atributo *boolean* chamado *crescente*, que por padrão é setado para *true* no construtor, fazendo a ordenação das listas de objetos direção ascendente.

Já que na nossa arquitetura já identificamos três *mappedsuperclass* que irão compor as aplicações, já foi possível também a nível de arquitetura criar os comparators para cada uma das superclasses: *IdBeanComparator*, *AtivoBeanComparator* e *DescricaoBeanComparator*.

- **Pacote das interfaces (br.com.flopes.arquitetura.business.entity.interfaces)**

Como geralmente em todas as aplicações em java, a arquitetura faz uso de interfaces para garantir a implementações de métodos importantes, criando “um contrato” com as classes que as implementam. Na nossa arquitetura temos as seguintes interfaces:

- *IdBean*: apenas com os método *getId()* e *setId()* para uso das entidades;
- *AtivoBean*: apenas com os métodos *isAtivo()* e *setAtivo()*;
- *DescricaoBean*: apenas com os métodos *getDescricao()* e *setDescricao()*;
- *IdAtivoBean*: uma interface que estende as interfaces *IdBean* e *AtivoBean*;
- *IdAtivoDescricaoBean*: interface que estende de *IdAtivoBean* e *DescricaoBean*;

Com a descrição deste pacote, fechamos os pacotes que dão suporte às entidades e podemos montar o seguinte diagrama para exemplificá-lo:

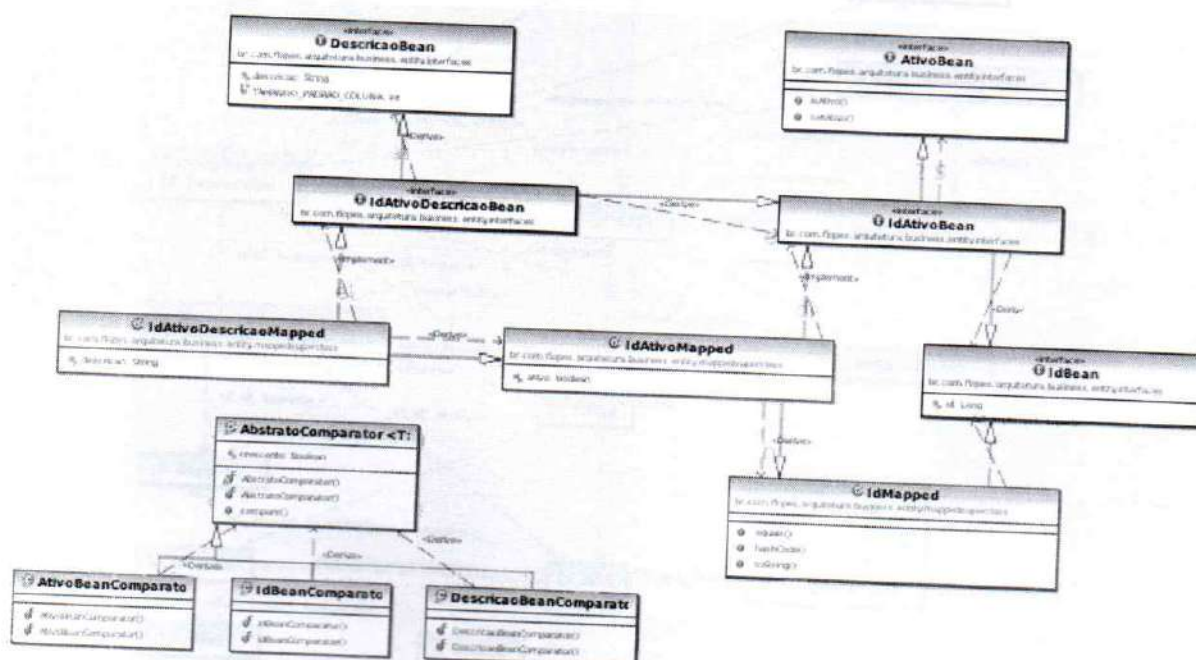


Figura 19 - Diagrama das classes e interfaces que darão suporte às entidades

Fonte: Resultado da pesquisa do autor

- **Pacote de persistência (br.com.flopes.arquitetura.persistence)**

O nome deste pacote já bem intuitivo, nele se encontra as classes que nos darão suporte a

todas as ações CRUD do sistema. Este pacote contém apenas três classes:

- `PersistenceException`: classe responsável por gerar a exceção do tipo `PersistenceException`;
- `Repository`: classe que contém alguns métodos abstratos de persistência e que é responsável pela instância do `entityManager`;
- `RepositoryImpl`: enquanto a classe `repository` faz a persistência de objetos no banco, a `RepositoryImpl` que estende `Repository` é responsável por criar uma estrutura de select compatível com o JPQL, nos dando uma maior liberdade em criar nossos selects trabalhando apenas com os objetos.

• Pacote de `persistencequery` (`br.com.flopes.arquitetura.persistence.query`)

Este pacote é um dos mais importantes que a arquitetura possui, pois para cada expressão que podemos montar em um JPQL existe um classe que tem seu papel bem definido. Neste pacote está contidas dezesseis classes, entre elas está a classe `Filter` que é responsável por criar as cláusulas *where*, `Join` para fazer junções de tabelas de várias formas, `left join`, `outer join`, etc, entre outras classes. Segue abaixo o diagrama de classe exemplificando as relações entre elas:

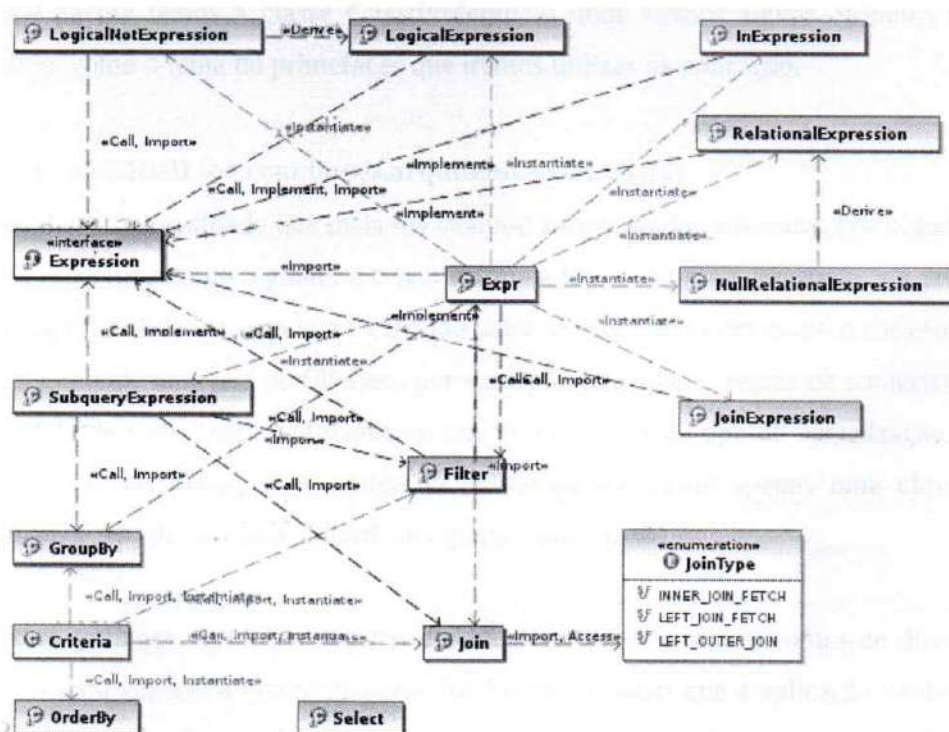


Figura 20 - Diagrama das classes e interface do pacote persistence query

Fonte: Resultado de pesquisa do proprio autor

- **Pacote do Spring (br.com.flopes.arquitetura.commons.springframework)**

Como o próprio nome diz, neste pacote está implementadas algumas classes que utilizam o framework spring. Dentre elas existem duas que se destacam: a ViewScope que é uma classe onde foi implementado um scope customizado, uma vez que o spring não disponibiliza este scope (até a versão de quando este trabalho foi escrito). Para não torna a arquitetura tão dependente de uma framework, foi criada a classe CdiScopeMetadaResolver que funciona como um adaptador para o spring dar suporte às anotações e contexto do CDI e JSF.

- **Pacote de segurança (br.com.flopes.arquitetura.security)**

Neste pacote estão todas as classes de segurança do sistema, acesso ao sistema via autenticação, permissões de acesso a módulos específicos, adição de permissões a grupos ou a usuários, permissões por transação, etc.

- **Pacote de view (br.com.flopes.arquitetura.view)**

Outro framework que a arquitetura dará suporte na camada de visão será o primefaces, uma framework de componentes ricos web que nos dá agilidade no desenvolvimento de telas. Neste pacote temos a classe GuestPreferences onde termos alguns atributos setados por padrão, como o tema do primefaces que iremos utilizar na aplicação.

- **Pacote de CRUD (br.com.flopes.arquitetura.view.crud)**

Uma das funcionalidade que mais me chamou a atenção dos softwares que o google produz, mais especificamente o gmail, é o fato de que a tela é sempre a mesma e, via javascript, faz um jogo de renderiza uma tela e esconde outra. A arquitetura em questão dará suporte a algo parecido onde uma tela de Clientes, por exemplo, terá apenas regras de renderizar uma lista, uma tela de visualização de dados ou um formulário. Este tipo de visualização ficará mais claro com um exemplo de aplicação. Neste pacote existe apenas uma classe chamada Visualização que nos dará suporte ao recurso comentado.

Nessa primeira versão da arquitetura, foi desenvolvida esta estrutura de diretório que, a princípio, nos dará suporte a grande maioria das funções básicas que a aplicação venha a precisar. Como podemos ver na figura abaixo, é uma estrutura ainda simples e em desenvolvimento que tenderá a crescer assim que for utilizada em mais sistemas.

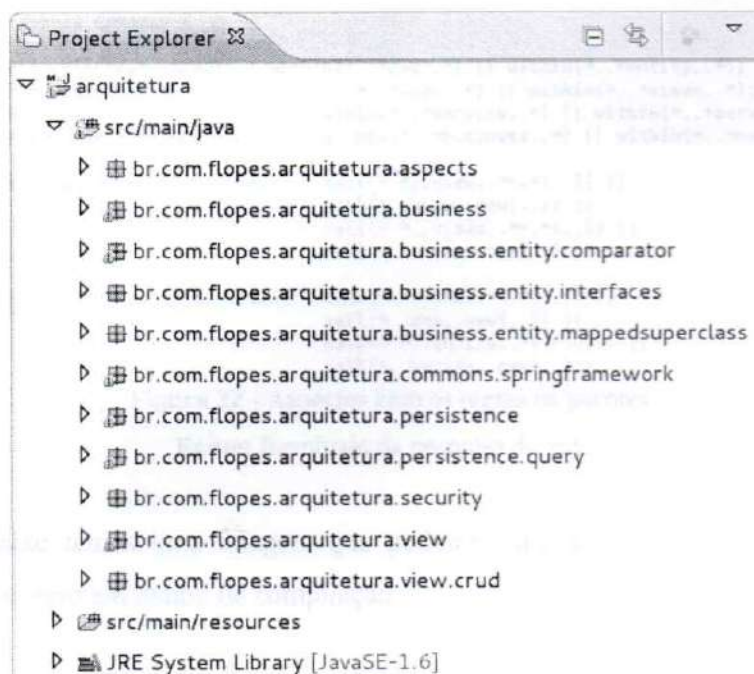


Figura 21 - Estrutura de pacotes da arquitetura

Fonte: Resultado da pesquisa do autor

4.1.2. Aspectos arquiteturais

A um nível de arquitetura foi pensado em desenvolver aspectos que analise tanto a arquitetura quanto a aplicação que irá utilizá-la. Sendo assim, foram definidos quais os problemas que o aspecto irá atacar:

- Regras para instanciação em camadas
- Regras de verificação de atributos
- Regras para não permitir o uso do *System.out*
- Regras para permissões de acesso
- Regras para logging de operações de persistência

Logo abaixo podemos analisar uma imagem onde foram criados os pointcuts que serão responsáveis por identificar de onde uma determinada chamada está vindo (com relação a pacotes) e também identificar qual o pacote da classe atual que estou trabalhando. Tendo essas informações podemos realizar algumas regras como, por exemplo, a camada de visualização não ter acesso a camada de persistência diretamente. Segue a imagem:

```

4 public aspect Camadas {
5
6     public pointcut inDAO() : within(*..dao..*) || within(*..entity..*);
7     public pointcut inView() : within(*..view..*) || within(*..visao..*);
8     public pointcut inService() : within(*..service..*) || within(*..servico..*);
9     public pointcut inBusiness() : within(*..business..*) || within(*..negocio..*);
10
11     public pointcut callToView() : call(* ..view..*.*(..)) ||
12                                     call(*..view..new(..)) ||
13                                     call(* ..visao..*.*(..)) ||
14                                     call(*..visao..new(..));
15
16     public pointcut callToDAO() : call(* ..dao..*.*(..)) ||
17                                     call(*..dao..new(..)) ||
18                                     call(* ..entity..*.*(..)) ||
19                                     call(*..entity..new(..));

```

Figura 22 - Aspectos com as regras de pacotes

Fonte: Resultado da pesquisa do autor

Logo abaixo temos uma imagem que podemos analisar como o aspecto age sobre as classes, gerando um erro em tempo de compilação.

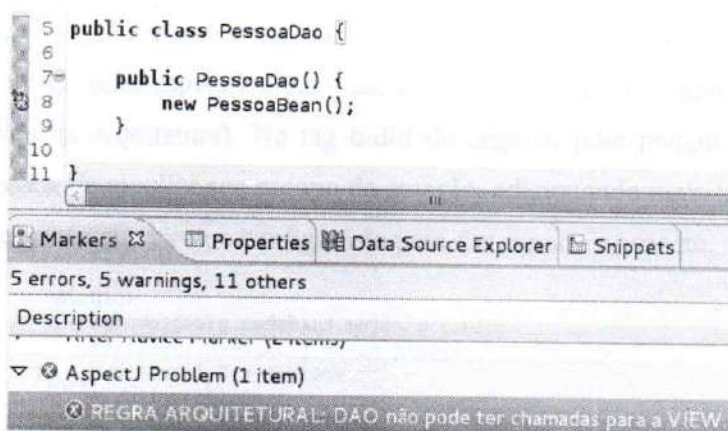


Figura 23 - Erro em tempo de compilação gerado pelo aspecto

Fonte: Resultado da pesquisa do autor

4.2. UTILIZANDO A ARQUITETURA NO DESENVOLVIMENTO DE UM SISTEMA

Neste trabalho utilizamos a tecnologia maven para que o ambiente de desenvolvimento seja padronizado. Para um sistema qualquer utilizar a arquitetura, esta deverá estar disponível no repositório do maven para que as aplicações consigam importá-la em suas dependências. No arquivo *pom* da arquitetura, a tag *packaging* deverá ser obrigatoriamente do tipo *jar* (`<packaging>jar</packaging>`).

Já no projeto que utilizará a arquitetura deverá declarar em suas dependências o arquivo *jar* gerado da arquitetura, incorporando assim quase todos os recursos nela disponível.

```

14<dependencies>
15  <dependency>
16    <groupId>br.com.flopes</groupId>
17    <artifactId>arquitetura</artifactId>
18    <version>1.0</version>
19    <type>jar</type>
20  </dependency>
21</dependencies>

```

Figura 24 - Declaração de dependência no arquivo POM

Fonte: Resultado da pesquisa do autor

Repare que falamos que quase todos os recursos da arquitetura estaria disponíveis, uma vez que os aspectos nela contidos não surgirão efeitos na aplicação. Para isso, vamos realizar uma configuração mais específica no maven.

4.2.1. Configuração maven

Para que a aplicação utilize de fato a arquitetura, além de declarar a dependência, temos que realizar uma configuração específica para que os aspectos consigam interagir com a aplicação (os aspectos contidos na arquitetura). Na tag `build` do arquivo `pom` podemos inserir plugins ao maven com a finalidade de ampliar seu escopo de atuação, adicionando mais funcionalidades a ele, e é juntamente neste ponto que iremos configurá-lo para dar suporte a aspecto.

```

<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>aspectj-maven-plugin</artifactId>
  <version>1.0</version>
  <configuration>
    <source>1.6</source>
    <target>1.6</target>
    <aspectLibraries>
      <aspectLibrary>
        <groupId>br.com.flopes</groupId>
        <artifactId>arquitetura</artifactId>
      </aspectLibrary>
    </aspectLibraries>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
        <goal>test-compile</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

Figura 25 - Adicionando um plugin ao maven

Fonte: Resultado da pesquisa do autor

4.2.2. Restrições na programação em camada

Ao realizar as configurações necessárias em um projeto java, adicionando a arquitetura nas

dependências e configurando o maven com o plugin do aspectj, este projeto estará pronto para seguir as regras definidas nos aspectos. Uma vez que as regras são infringidas, automaticamente a IDE mostra erro de compilação na tela, erro que deverá ser corrigido.

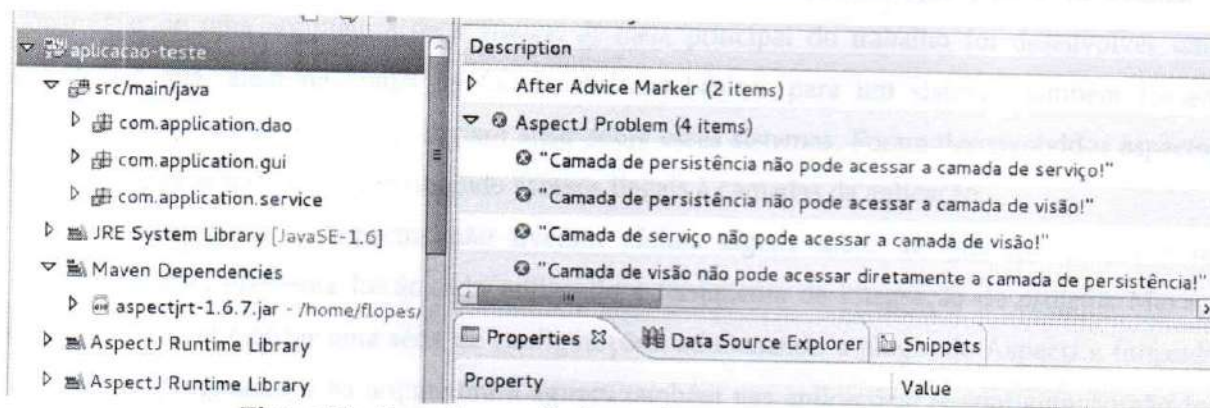


Figura 26 - Erros de compilação gerados pelas regras dos aspectos

Fonte: Resultado da pesquisa do autor

Como a configuração do maven foi realizada corretamente, a aplicação é válida em tempo de compilação pelos aspectos definidos na arquitetura. Para finalizar os testes com os aspectos vamos tentar compilar via maven (`mvn compile`) e verificar o que o terminal nos retorna.

```
[INFO] --- aspectj-maven-plugin:1.3:compile (default) @ architectural-enforcement ---
[ERROR] "Camada de visão não pode acessar diretamente a camada de persistência!"
[ERROR] "Camada de serviço não pode acessar a camada de visão!"
[ERROR] "Camada de persistência não pode acessar a camada de visão!"
[ERROR] "Camada de persistência não pode acessar a camada de serviço!"
[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time: 2.561s
[INFO] Finished at: Mon Dec 05 10:08:35 BRT 2011
[INFO] Final Memory: 5M/97M
[INFO] -----
[ERROR] Failed to execute goal org.codehaus.mojo:aspectj-maven-plugin:1.3:compile (defa
[ERROR] error at someDao.find();
[ERROR] ~~~~~
[ERROR] /home/flopes/workspace-pessoal/aplicacao-teste/src/main/java/com/application/gu
[ERROR] see also: /home/flopes/workspace-pessoal/aplicacao-teste/src/main/java/com/appl
[ERROR] error at someGui.render();
[ERROR] ~~~~~
```

Figura 27 - Erro de compilação ao tentar compilar utilizando os comando do maven

Fonte: Resultado da pesquisa do autor

O maven se comportou como esperado, barrando a compilação da aplicação e nos forçando a corrigir os problemas na aplicação.

5. CONCLUSÃO

Neste trabalho de conclusão de curso foi possível analisar alguns recursos que a programação orientada a aspectos nos oferece, focando nos recursos que pudessem auxiliar a construção de uma arquitetura de software. A ideia principal do trabalho foi desenvolver uma arquitetura que, além de conter as funcionalidades básicas para um sistema, também fossem desenvolvidos os aspectos que poderiam atuar sobre esses sistemas. Foram desenvolvidos aspectos para restrições arquiteturais, restringindo acessos ilegais à camadas da aplicação.

A princípio os aspectos não tiveram efeitos algum sobre as aplicações, apenas na arquitetura. Este problema foi atacado utilizando a ferramenta de integração de projetos Maven, onde foi possível realizar uma série de configurações, adicionando o plugin do AspectJ e forçando os aspectos programados na arquitetura a agirem também nas aplicações. A configuração não foi realizada de forma trivial, houve uma certa dificuldade em encontrar artigos e tutoriais que informassem como realizar essas configurações para a utilização do Maven com a biblioteca AspectJ. Ao final de vários teste, conseguimos chegar a uma configuração até que simple e que resolveu os nossos problemas.

Também foi possível verificar neste trabalho uma boa integração da POA com IDE's de programação como Eclipse e Netbeans. As duas ferramentas de desenvolvimento apresentaram um excelente suporte à aspectos tanto para criação das unidades modulares quanto à integração com a parte visual, identificando os erros gerados pelos aspectos e mostrando ao programador de uma forma bem clara.

Programação orientada a aspectos se mostrou bem eficiente no que se propõe a fazer, resolver os problemas de interesses transversais. Foi possível definir alguns pontos da aplicação em que se poderia utilizar aspectos. Foram também realizadas as suas implementações de uma forma satisfatória. A única observação negativa que podemos fazer com relação a utilização de aspectos é que, em excesso, uma aplicação pode ter sua compreensão comprometida, uma vez que classes e aspectos trabalham em paralelo e os aspectos podem exercer uma forte influência sobre essas classes, alterando seu comportamento sem que o desenvolvedor perceba.

Portanto, a utilização de aspectos é bem interessante no desenvolvimento de sistemas, mais é preciso definir bem quais pontos queremos tratar com aspectos e fazer o uso racional deste paradigma sem comprometer a compreensão da aplicação.

REFERÊNCIAS

- ACTIVEASPECT.** 2005. Último acesso em 27 de Setembro de 2011. Disponível em: <<http://www.cs.ubc.ca/labs/spl/projects/activeaspect/>>.
- AJDT. AspectJ Development Tools (AJDT).** 2007. Último acesso em 29 de Julho de 2011. Disponível em: <<http://www.eclipse.org/ajdt/>>.
- ALVIM, Paulo. Tirando o Máximo do JAVA EE 5 Open-Source com jCompany Developer Suite – Volume I – Fundamentos.** Ed. PowerlogicPublishing 2007.
- ASPECTJ. The AspectJ Programming Guide.** 2003. Último acesso de 17 de maio de 2011. Disponível em: <<http://www.eclipse.org/aspectj/doc/released/progguide/>>.
- CYSNEIROS, L. M. Requisitos Não Funcionais: Da Elicitação ao Modelo Conceitual.** 2001. 224 f. Tese (Doutorado em Ciências da Computação) - Departamento de Informática, PUC/RJ, Rio de Janeiro. 2001.
- ECLIPSE. Eclipse IDE Project.** 2007. Último acesso em 21 de Janeiro de 2011. Disponível em: <<http://eclipse.org>>.
- GRADECKI, J. D; LESIECKI, N. Mastering AspectJ: Aspect-Oriented Programming in Java.** Indianapolis: Wiley Publishing, Inc., 2003.
- MARTINS, J. C. C. Técnicas para Gerenciamento de Projetos de Software.** Rio de Janeiro: Brasport. 2007.
- MOREIRA, Gustavo. O que é o maven e seus primeiros passos com a ferramenta.** 2010. Último acesso em 30 de Agosto de 2011. Disponível em: <http://blog.dclick.com.br/2010/09/15/o-que-e-o-maven-e-seus-primeiros-passos-com-a-ferramenta>
- NELSON, Maria A. V., NELSON, Torsten, COWAN, Donald, ALENCAR, Paulo. An Approach to Formal Automated Analysis of Problem-Frame Concerns.** Aceito para publicação em Information and Software Technology, 2005.

NELSON, Torsten, COWAN, Donald, ALENCAR, Paulo. **Formal Verification of a Bounded Buffer with Three Separate Concerns**. In **proceedings of the Workshop on Advanced Separation of Concerns in Software Engineering, at the International Conference on Software Engineering - ICSE**, Toronto, Canada, May 2001.

KICZALES, G. et al. **An overview of AspectJ**. *Lecture Notes in Computer Science*, 2001, 327-355. Disponível em: <citeseer.ist.psu.edu/kiczales01overview.html>.

KICZALES, G. et al. **Aspect-oriented programming**. In: AKSIT, M.; MATSUOKA, S. (Ed.). **Proceedings European Conference on Object-Oriented Programming**. Berlin, Heidelberg, and New York: Springer-Verlag, 1997. v. 1241, p. 220-242. Disponível em: <citeseer.ist.psu.edu/kiczales97aspectoriented.html>.

KICZALES, G.; MEZINI, M. **Aspect-oriented programming and modular reasoning**. In: ICSE '05: **Proceedings of the 27th international conference on Software engineering**. [S.l.: s.n.], 2005. p. 49-58. ISBN 1-59593-963-2.

OSSHHER, H.; TARR, P. L. **Operation-level composition: A case in (join) point**. In: ECOOP '98: **Workshop on Object-Oriented Technology**. London, UK: Springer-Verlag, 1998. p. 406-409. ISBN 3-540-65460-7.

WINCK, Diogo Vinícius. **AspectJ: programação orientada a aspectos com java** / Diogo Vinícius Winck, Vicente Goetten Junior ; São Paulo : Novatec Editora, 2006.

WINCK, Diogo Vinícius, GOETTEN, Vicente. **AspectJ em 20 minutos**. Último acesso em 12 de Agosto de 2011. Disponível em: <http://javafree.uol.com.br/artigo/871488/AspectJ-em-20-minutos.html>