UNIVERSIDADE FEDERAL DE PERNAMBUCO
GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO
CENTRO DE INFORMÁTICA

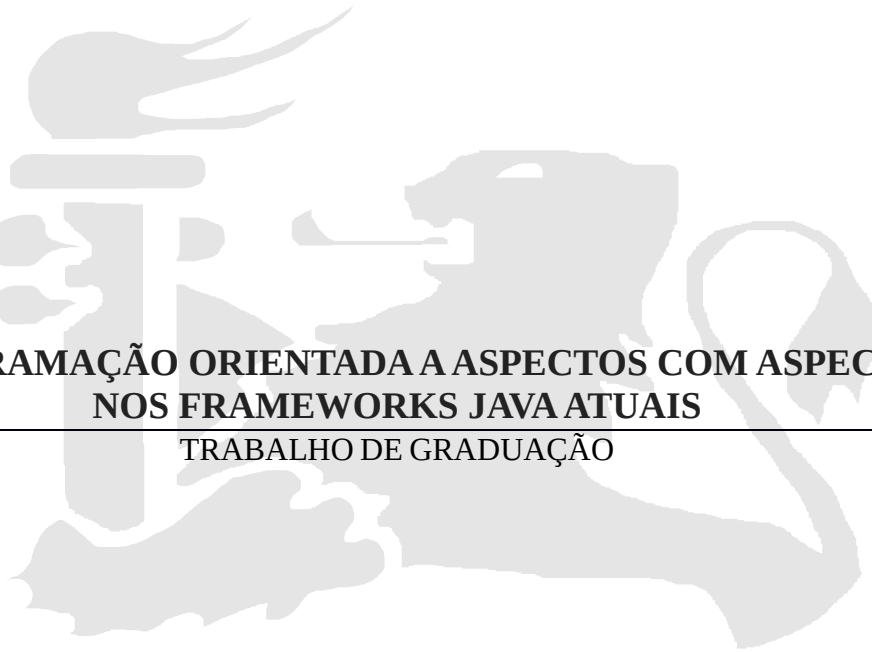
FAGNER FERNANDES CANDIDO DA SILVA

Programação Orientada a Aspectos com AspectJ nos frameworks Java atuais

TRABALHO DE GRADUAÇÃO

RECIFE
2016

FAGNER FERNANDES CANDIDO DA SILVA



PROGRAMAÇÃO ORIENTADA A ASPECTOS COM ASPECTJ NOS FRAMEWORKS JAVA ATUAIS

TRABALHO DE GRADUAÇÃO

Trabalho de Graduação em Sistemas de Informação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: *Prof. Henrique Emanuel Mostaert Rebêlo*

RECIFE
2016

FAGNER FERNANDES CANDIDO DA SILVA

PROGRAMAÇÃO ORIENTADA A ASPECTOS COM ASPECTJ NOS FRAMEWORKS JAVA ATUAIS

TRABALHO DE GRADUAÇÃO

Trabalho de Graduação apresentado à graduação em Sistemas de Informação do Centro de Informática da Universidade Federal de Pernambuco para obtenção do grau de Bacharel em Sistemas de Informação.

Recife, _____ de janeiro de 2016.

BANCA EXAMINADORA

_____

Prof.º Henrique Emanuel Mostaert Rêbelo
(Orientador)

_____

Prof.º Fernando José Castor de Lima Filho
(Avaliador)

Agradecimentos

Agradeço, primeiramente, a Deus por ter me dado forças e coragem para chegar até aqui.

A toda minha família, mãe e pai, por todo suporte dado durante a juventude sem o qual não seria possível tornar possível mais essa conquista.

A minha esposa Rafaela, que teve uma paciência extrema e apoiou bastante desde início deste curso e não via a hora que conseguisse concluí-lo.

A minha filha Isadora, que sempre me recebeu com sorriso enorme ao chegar em casa e foi a minha fonte de inspiração ao longo dessa jornada.

Ao TRF, minha supervisora Gleicy D'Lyzandra e meu diretor Arnaldo Pereira pela flexibilização do horário no último semestre, sem o qual não seria possível concluir do curso.

Ao Centro de Informática (CIn) da UFPE, pois foi um ambiente único no qual pude ter contato com os mais diversos professores que me incentivaram a continuar nos estudos apesar de todas as dificuldades.

Ao Prof. Henrique, por seu grande apoio e orientação no decorrer deste trabalho.

Por fim, aos meus amigos que de alguma forma contribuíram para realização deste trabalho. Bem como, aqueles outros que se dispuseram a me ajudar e se colocaram à disposição.

Resumo

O processo de desenvolvimento de software vem evoluindo com passar do tempo. Isso se deve a evolução e adaptação desse processo assim como dos modelos e metodologias relacionadas a engenharia de software. Por outro lado, os projetos de software estão se tornando cada vez mais complexos e se espera que essa complexidade aumente ainda mais com o surgimento de novas tecnologias e a necessidade de as integrar. No cenário de desenvolvimento web atual, existem vários frameworks Java, como JSF e Hibernate que auxiliam nesse processo de desenvolvimento. Outra técnica que buscar reduzir essa complexidade é a programação orientada a aspectos. Sendo assim, é preciso saber se o uso desse paradigma ainda é relevante no cenário de desenvolvimento web atual. Para isso, foram construídas duas versões do sistema Health Watcher: a primeira, usando a orientação a objetos; e a segunda, usando aspectos com uso do AspectJ. Dessa forma, o objetivo desse estudo é identificar quais são os benefícios e as limitações do uso do paradigma orientado a aspectos para o desenvolvimento de aplicações na plataforma Java Enterprise Edition.

Palavras-chave: AspectJ, Java, Hibernate, JSF, Programação orientada a aspectos.

Abstract

The software development process has evolved over time. This is due to evolution and adaptation of software development process as well as the models and methodologies related to software engineering. On the other hand, software projects are becoming increasingly complex and this complexity is expected to increase even further with the emergence of new technologies and the need to integrate them. In the current web development scenario, there are several Java frameworks such as JSF and Hibernate that assist development process. Another technique that seek to reduce complexity is the aspect-oriented programming. Therefore, it is necessary to know whether the use of this paradigm is still relevant in today's web development scenario. For this, two versions of the Health Watcher system were built: the first, using object orientation; and second, using aspects with the use of aspectj. Thus, the aim of this study is to identify the benefits and limitations of using Aspect-oriented paradigm for application development in the Java Enterprise Edition platform.

Keywords: AspectJ, Java, Hibernate, JSF, Aspect-Oriented Programming.

Lista de Figuras

Figura 1 - Evolução da plataforma JEE 11
Figura 2 - Arquitetura JSF 17
Figura 3 - Ciclo de Vida do JSF 19
Figura 4 - Tempo de Resposta do Ciclo de Vida 21
Figura 5 - Entidade Employee 22
Figura 6 - Tabela Employee no PostgreSQL 23
Figura 7 - Papel do Hibernate 24
Figura 8 - Composição de um sistema 25
Figura 9 - Entrelaçamento 26
Figura 10 - Interesses numa única dimensão 27
Figura 11 - Espalhamento 27
Figura 12 - Pointcut 29
Figura 13 – Advice 30
Figura 14 - Arquitetura HW versão original OO 32
Figura 15 - Arquitetura HW Nova Versão OO 33
Figura 16 – Métricas SoC 36
Figura 17 - Exemplo Named Query 38
Figura 18 - Named Query HW OO 38
Figura 19 - Named Query HW AO 39
Figura 20 - Anotação @JoinColumn HW OO 40
Figura 21 - @JoinColumn HW AO 40
Figura 22 - Problema no @JoinColumn HW AO 40
Figura 23 - Sequence Generator 41

Lista de Tabelas

Tabela 1 - Lista das J2EE 1.2 Especificações..... 14
Tabela 2 - Frameworks Java Atuais..... 15
Tabela 3 - Escopo no JSF..... 20
Tabela 4 - Resumo das API's do JPA..... 23
Tabela 5 - Join Points..... 29
Tabela 6 – Métricas..... 34
Tabela 7 - Casos de Uso..... 35
Tabela 8 - Valores coletados para as métricas por Separação de Interesses . 36
Tabela 9 - Valores coletados para as métricas por Tamanho..... 37

Sumário

1. Introdução 10
   1.1 Motivação 10
   1.2 Objetivos e Contribuições 11
   1.3 Estrutura do Documento 12
2. Contextualização 13
   2.1 Plataforma JEE 13
      2.1.1 História do Java Enterprise 13
   2.2 Frameworks Java atuais 15
      2.1.1 JSF 16
      2.2.2 Hibernate 21
   2.3 Programação orientada a aspectos 25
      2.3.1 Entrelaçamento 26
      2.3.2 Espalhamento 27
      2.3.3 AspectJ 28
      2.3.4 AspectJ Development Tools (AJDT) 30
3. Health Watcher 31
   3.1 Design Orientado a Objetos na versão original 31
   3.2 Design Orientado a Objetos na nova versão 32
   3.3 Design Orientado a Aspectos na nova versão 33
4. Analisando o AspectJ nas versões do sistema Health Watcher 34
   4.1 Procedimentos para avaliação 35
   4.2 Comparação de resultados 35
      4.2.1 Named Query 37
      4.2.2 Mapeamento Objeto-Relacional 39
5. Conclusão e Trabalhos Futuros 42
6. Referências 44

1. Introdução

Este capítulo apresentará a motivação para o desenvolvimento do projeto. Além disso, será descrito os objetivos deste trabalho, assim como a estruturação desse documento.

1.1 Motivação

O processo de desenvolvimento de software vem sofrendo drásticas mudanças ao longo das últimas décadas. Essas mudanças são consequência da evolução não só, do processo de criação e adaptação de produtos de software [1], como também dos modelos e das metodologias utilizadas na engenharia de software. Os projetos de software estão cada mais complexos e a tendência é que essa complexidade aumente nos próximos anos [2].

Uma forma de lidar com essa complexidade crescente é a modularização dos sistemas. Quando se divide um problema em partes menores, a concepção da solução ocorre de modo mais rápido e eficiente. Entretanto existem interesses (*concerns*) que afetam as funcionalidades do sistema em vários níveis [2].

Segurança, logging, pooling de recursos, caching, monitoramento de performance, concorrência e gerenciamento de transações são exemplos de funcionalidades que afetam vários módulos do sistema, sendo considerados assim características transversais (do inglês, *cross-cutting*).

Em 1996, foi concebida uma nova técnica de programação chamada programação orientada a aspectos (do inglês, aspect-oriented programming) [3] com intuito de solucionar esse problema cujo qual o paradigma orientado a objetos não pode resolver. Esse paradigma de programação orientada a aspectos é considerado ideal para esse tipo de separação, pois ele introduz uma nova unidade de modularização chamada de aspecto, onde cada aspecto pode ser responsável por cada uma dessas funcionalidades transversais.

Por outro lado, com a introdução da plataforma Java Enterprise Edition (JEE) 5, em 2006, e com lançamento da versão Java EE 6, em 2009, o desenvolvimento web atual foi bastante simplificado [4]. Com uma série de interfaces de Programação de Aplicação (do inglês, *Application Programming Interface*), a plataforma JEE elimina a necessidade de ter que implementar

serviços infraestruturais como controle de transações, threading ou monitoramento da aplicação [4]. E graças à Requisição de Especificação Java 299 (do inglês, *Java Specification Request 299*) foi criado o CDI (do inglês, *Contexts and Dependendy Injection*) permitiu que várias dessas funcionalidades transversais pudessem ser interligadas [4]. Portanto é possível dividir essas funcionalidades através do uso dessas API's do JEE. A figura 1 mostra a evolução da plataforma JEE ao longo do tempo. Podemos perceber que conforme essa plataforma foi evoluindo mais funcionalidades foram sendo adicionadas a esse ambiente.

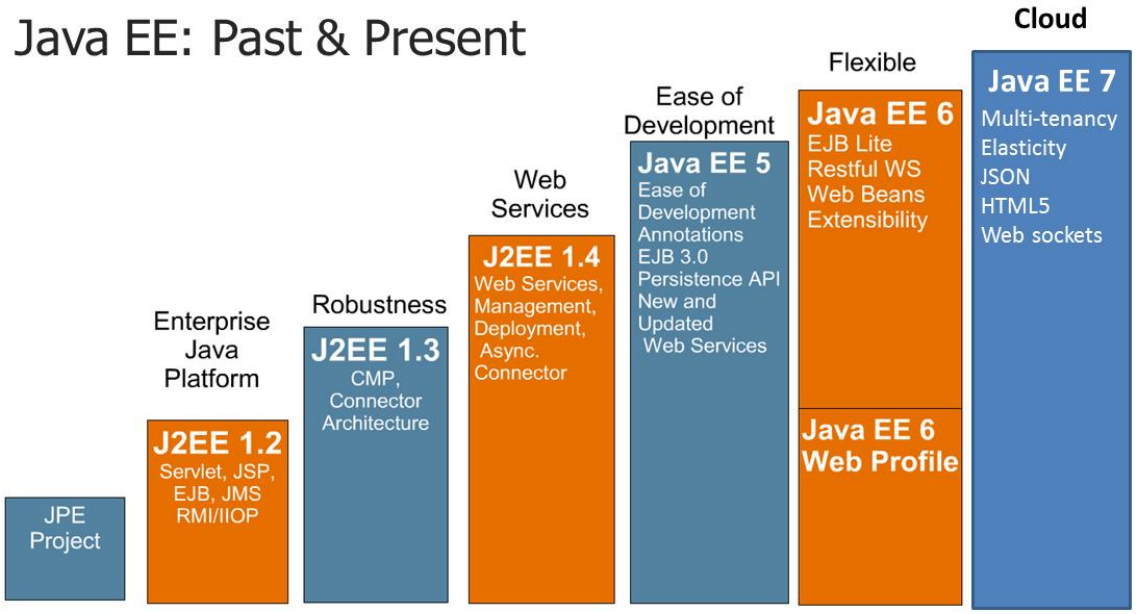


Figura 1 - Evolução da plataforma JEE

Sendo assim, com base no cenário atual é necessário avaliar se o uso do paradigma de programação orientada a aspectos ainda é relevante no cenário de desenvolvimento web atual.

1.2 Objetivos e Contribuições

Este trabalho de graduação tem como objetivo geral avaliar a viabilidade do uso da programação orientada a aspectos, mais especificamente, através da linguagem de programação, AspectJ tendo como base o cenário web atual. Para que isso concretize será utilizada uma aplicação chamada Health Watcher [39]. Após a migração dessa aplicação para plataforma java web atual, será feita construída uma nova versão desse software através do AspectJ. Como objetivos específicos deste trabalho, destacamos as seguintes atividades:

i. Identificar quais funcionalidades transversais podem ser transformadas em aspectos tendo como base um projeto desenvolvido sob o prisma dos frameworks Java atuais;
ii. Identificar os benefícios do uso da programação orientada a Aspectos tendo como base um projeto desenvolvido sob o prisma dos frameworks Java atuais;
iii. Comparar as versões do Health Watcher através de uma suíte de métricas para separação de interesses e tamanho [43, 44] das implementações do sistema.

A respeito das contribuições para esse trabalho, podemos destacar os seguintes itens:

- Analisar a forma como os interesses transversais podem tratados na plataforma JEE;
- Verificar se a programação orientada a aspectos oferece algum tipo de benefícios no cenário de desenvolvimento web atual;
- Destacar os benefícios e as desvantagens do uso da programação orientada a aspectos no cenário atual JEE;

1.3 Estrutura do Documento

Os capítulos seguintes estão estruturados da seguinte forma: O capítulo 2 aborda a contextualização deste trabalho, apresentando a composição da plataforma JEE atual e suas versões o longo do tempo, assim como os frameworks Java atuais e os conceitos de programação orientada a aspectos (AOP) e a linguagem de Programação AspectJ. O capítulo 3 detalha a estrutura do sistema Health Watcher e descreve os casos de uso implementados nesse sistema. O capítulo 4 trata dos benefícios e das desvantagens existentes em cada abordagem de desenvolvimento. Por fim, o capítulo 5 expõe a conclusão deste trabalho, além de sugestões para projetos futuros.

2. Contextualização

Nesse capítulo serão abordados os conceitos principais que permeiam esse trabalho. Na seção 2.1, abordaremos sobre a plataforma Java Enterprise Edition (JEE), falando um pouco da sua história na seção 2.1.1 e as suas especificações na seção 2.1.2. Mais adiante, iremos explicar brevemente na seção 2.2 sobre a multiplicidade de frameworks existentes no cenário web do Java atual na qual será dado um destaque a dois deles: o Java Server Faces (JSF) na seção 2.2.1 e o Hibernate na seção 2.2.2. Na seção 2.3, apresenta o conceito da programação orientada a aspectos (POA) e que tipos de problemas ela se propõe a resolver (como entrelaçamento e espalhamento, nas seções 2.3.1 e 2.3.2, respectivamente), assim como discorre sobre o AspectJ, a linguagem de programação que implementa os conceitos existentes na POA para o Java na seção 2.3.3 e o AJDT na seção 2.3.4.

2.1 Plataforma JEE

A Java EE (do inglês, *Java Plataform, Enterprise Edition*) é uma plataforma padrão para desenvolver aplicações Java de grande porte e/ou para internet, que inclui bibliotecas e funcionalidades para implementar software Java de forma distribuída [20] utilizando do padrão cliente-servidor.
A seguir, vamos detalhar sobre como surgiu a plataforma JEE e o que cada uma das suas versões trouxe de novidade.

2.1.1 História do Java Enterprise

Inicialmente, a Sun estava focada no desenvolvimento do kit de desenvolvimento Java (do inglês, *Java Development Kit*) [22] e não havia planos para criação de uma versão voltada para as empresas.

Entretanto, a Sun já tinha ciência da necessidade de ferramentas mais avançadas para o desenvolvimento de aplicações, particularmente por conta do crescimento da Internet e a popularidade das aplicações web [35].

Sendo assim, em maio de 1998, foi lançado o projeto Java Professional Edition (JPE) pela Sun [23, 24, 35]. Em 1999, foi lançada a J2EE 1.2 (do inglês, *Java 2 Plataform, Enterprise Edition*). Essa versão trouxe 10 especificações conforme mostra a tabela 1.

Tabela 1 - Lista das J2EE 1.2 Especificações

Tecnologia	Versão
JDBC Standard Extension API	2.0
Java Naming and Directory Interface Specification (JNDI)	1.2
RMI-IIOP	1.1
Java Servlet	2.2
JavaServer Pages (JSP)	1.1
Enterprise JavaBeans (EJB)	1.1
Java Message Services (JMS)	1.0
Java Transaction API (JTA)	1.0
JavaMail API	1.1
JavaBeans Activation Framework (JAF)	1.0

A versão 1.3 foi lançada em 2001 e trouxe como novidades as seguintes especificações: JCA, JAPX e JAAS que tratam, respectivamente, sobre – conectividade entre sistemas corporativos e servidor de aplicação; processamento de arquivos XML e autenticação e autorização de usuários.

A versão 1.4 foi lançada em 2003 e passou a dividir as especificações por categorias: tecnologias relacionadas a web services, tecnologias relacionadas a aplicação web, tecnologias relacionadas a aplicação corporativa e tecnologias relacionadas a gerenciamento e segurança.

Em resposta as reclamações dos desenvolvedores sobre o tamanho das versões anteriores, o J2EE foi rebatizado como Java EE e essa versão 5 foi lançada em 2006 trazendo mais especificações que foram adicionadas através de JSR (*Java Specification Requests*) específicas.

A versão 6 introduziu o conceito de perfil que representa a configuração da plataforma de acordo com a necessidade do usuário. Além disso, ela começou o processo de remoção das API's antigas e que caíram em desuso.

Em junho de 2013, foi lançada a versão 7 e acompanhando a popularidade do HTML adicionou novas API's para WebSocket, processamento JSON, Servlet Assíncrono e NIO não bloqueante (do inglês, *non-blocking NIO*). Além disso, foi adicionada uma solução que permite o processamento em batch (em lote).

O lançamento da versão 8 está previsto para final de 2016 e início de 2017. O objetivo principal dessa versão é o suporte para HTML5 e padrão HTTP 2.0, além de melhorar a infraestrutura de aplicações que rodam na nuvem.

Podemos perceber que a cada versão do JEE, mais tecnologias estão sendo adicionadas só que dessa vez é possível selecionar apenas aquelas que são necessárias de acordo com a realidade do projeto

Na seção 2.2 iremos falar sobre alguns dos frameworks Java atuais, dando uma maior atenção ao JSF e Hibernate.

2.2 Frameworks Java atuais

Com relação ao termo framework, nem sempre compreendemos o significado dessa palavra. Um framework é essencialmente o esqueleto reusável de uma linha de produto de software implementada integralmente numa linguagem de programação orientada a objetos [5]. Ou seja, é uma coleção de artefatos de software que possuem uma interdependência entre si e possuem uma finalidade específica. Além disso, ele pode ser estendido a fim de inserir novas funcionalidades. A tabela 2 mostra uma coleção desses frameworks mais utilizados para plataforma enterprise Java [6]:

Tabela 2 - Frameworks Java Atuais

Nome	Funcionalidade	Publicação	Versão Atual
Spring	Inversão de	2002	4.2.2
CDI	Inversão de	2013	1.1
EJB	Inversão de	1998	3.2
Hibernate	Persistência	2001	5.0.6
JavaServer Faces	UI	2004	2.2.5
JUnit	Testes	2000	4.12
Apache Log4J 2	Logging	2012	2.5

Existem uma grande variedade de frameworks disponíveis, mas optamos por enumerar esses sete frameworks devido seu grande uso no desenvolvimento web atual [36]. A seguir, vamos detalhar mais sobre o JSF e sua arquitetura assim como Hibernate.

2.1.1 JSF

JavaServer Faces (JSF) é um novo framework padrão Java para construção de aplicações web [7, 18]. Ele simplifica o desenvolvimento provendo uma abordagem centrada em componentes para desenvolvedor. Esses componentes podem ser reutilizados e fazem parte da composição de interfaces para o usuário. E como ele é um padrão Java, ele pode ser expandido por vendedores de ferramentas de desenvolvimento como JBoss (RichFaces) e Primefaces (PrimeUI). Outro aspecto importante do JSF é que ele é baseado no servidor (do inglês, *serverbased*), o que significa que os seus componentes e seus respectivos estados são representados no servidor com um ciclo de vida bem definido [11]. Para que uma aplicação JSF seja executada é preciso que ela esteja dentro de um contêiner web como Tomcat ou Jetty, por exemplo.

2.1.1.1 História e versões

O JSF teve sua primeira versão lançada em 2004 [13, 14]. Ela possuía uma elevada quantidade de bugs e não era considerada madura o suficiente para ser usada em larga escala. Ainda em 2004, foi lançada a versão 1.1 que tinha como objetivo corrigir os bugs reportados na versão anterior. Em 2006, foi lançada a versão 1.2 que veio com uma série de melhorias e foi bem aceito pela comunidade de desenvolvedores. Em 2009, foi lançada a versão 2.0 que trouxe várias mudanças técnicas e funcionais [14]. Entre elas, as mais marcantes foram: a possibilidade de se utilizar templates através do Facelets (o que tornou a modularização da interface gráfica ainda mais fácil) e a inclusão de um novo escopo chamado escopo de visão (do inglês, *view scope*), o que permitiu o uso do escopo de conversação de forma nativa. Anteriormente, isso era feito de forma não padronizada, o que não facilitava a manutibilidade.

Em 2010, foi lançada a versão 2.1 que não trouxe grandes melhorias em relação a versão anterior [15]. Em 2013, foi lançada a versão 2.2 que trouxe suporte a multi-tenancy (múltiplos inquilinos em tradução livre). Multi-tenancy é um padrão arquitetural na qual uma única instância do software está rodando na infraestrutura do provedor de serviço e vários inquilinos acessam a mesma instância [16]. Ou seja, é uma abordagem bastante utilizada no contexto da

computação em nuvem, mais especificamente no modelo de Software como Serviço (do inglês, Software as a Service).

Atualmente, a versão 2.3 está prevista para 2017 [17]. Nessa versão estão previstas uma série de novas características que se relacionam com outras *features* da plataforma JEE como: alinhamento com CDI, Ciclo de Vida do JSF, Conversão e Validação, segurança de tipos (do inglês, *type-safety*), dentre outros.

2.1.1.2 Arquitetura JSF

Sua arquitetura é baseada no padrão de design (do inglês, *design pattern*) MVC (Model-View-Controller) [7,8]. O propósito desse padrão é desacoplar os dados da camada de apresentação [9], o que permite uma clara separação de responsabilidades entre os elementos que compõem uma aplicação JSF como mostra a figura 2.

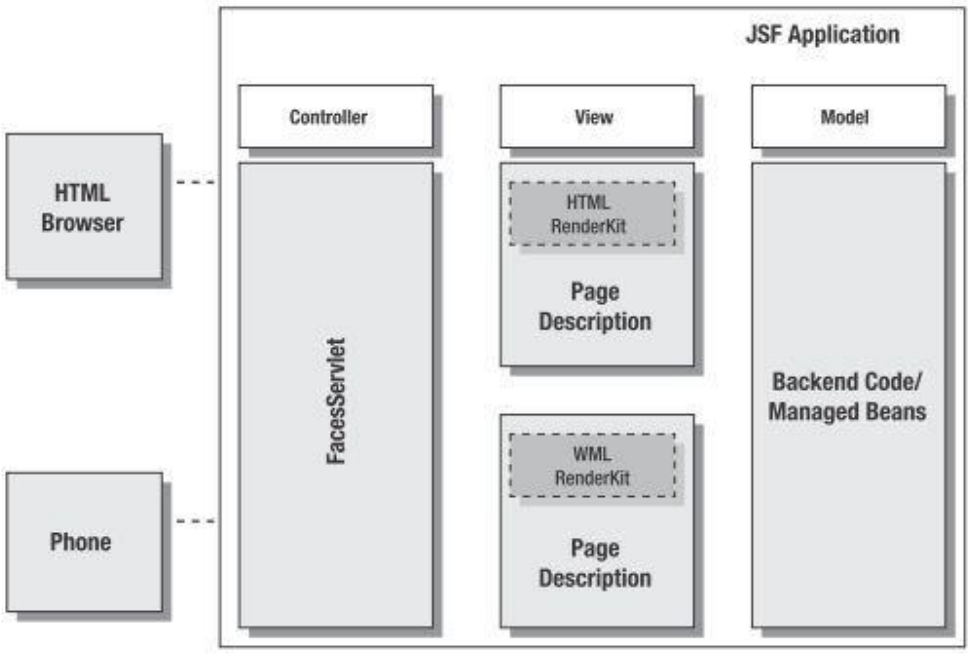


Figura 2 - Arquitetura JSF

Fazendo uma breve explanação sobre cada elemento do padrão MVC, o FacesServlet exerce a função de Controller sendo ele responsável por controlar o fluxo da navegação e disparar as requisições para as páginas apropriadas [8]. Já os componentes da UI (do inglês, *user interface*) fazem a função da

View. E por último, o bean gerenciado (do inglês, *managed bean*) exerce a função de Model onde ele é responsável pela lógica da aplicação e pode ser registrado num arquivo de configuração chamado faces-config.xml [8] quanto através de anotações [10].

2.1.1.3 Ciclo de vida do JSF

Quando se fala em ciclo de vida de uma aplicação se refere a vários estágios do processamento de uma aplicação [19]. O ciclo de vida do JSF (do inglês, *JSF lifecycle*) começa quando o cliente faz uma requisição HTTP para uma página e acaba quando o servidor responde com a página traduzida em HTML. Esse processo pode ser divido em duas fases: execução e renderização. A primeira fase é dividida ainda em subfases que suportam uma sofisticada árvore de componentes. Essa estrutura requer que os dados dos componentes sejam convertidos e validados, os eventos dos componentes sejam lidados e os dados dos componentes sejam propagados aos beans (também chamados de JavaBeans que são programas de software reusáveis que podem ser desenvolvidos e reunidos facilmente para criar aplicações sofisticadas [20]) de forma ordenada.

Uma página JSF é representada por uma árvore de componentes chamada de visão (do inglês, *view*). Durante o ciclo de vida, a implementação do JSF deve construir a visão enquanto deve levar em consideração o estado salvo da requisição anterior da página. Quando o cliente faz uma requisição, a implementação faz várias tarefas, como validação dos dados inseridos nos componentes na visão e conversão dos dados em tipos especificados no lado do servidor. Essas tarefas estão representadas na figura 3.

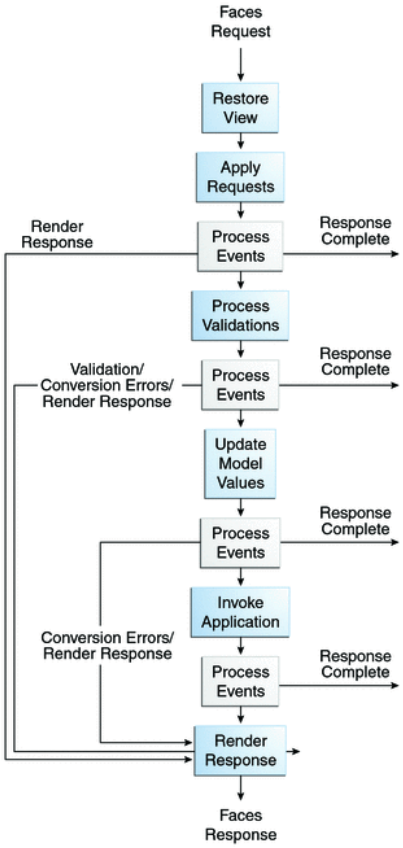


Figura 3 - Ciclo de Vida do JSF

Na fase de Restore View, o JSF constrói a visão da página, atrela os tratadores de eventos e validadores aos componentes e salva a visão na instância do FacesContext.

Na fase de Apply Request, os valores dos componentes são atualizados através de parâmetros enviados na requisição. Ao final dessa fase, as mensagens e os eventos são enfileirados.

Na fase de Process Validations, todos os validadores registrados na árvore de componentes verificam se os valores estão em acordo as regras de cada validador. Caso haja algum erro de validação ou conversão, esse erro é adicionado a instância do FacesContext e o ciclo de vida avança para a fase de Render Response.

Na fase de Update Model Values, depois que JSF verifica que os dados são válidos, eles são enviados para o objeto no lado do servidor. Caso ocorra algum erro nessa fase o ciclo de vida avança para a fase de Render Response.

Na fase de Invoke Application, o JSF lida com quaisquer eventos no nível da aplicação, como por exemplo, a submissão de um formulário ou ligação com outra página.

E por fim, na fase de Render Response, o JSF constrói a visão e delega a autoridade para recurso apropriado para renderizar as páginas. Além disso, caso tenha ocorrido algum erro nas fases anteriores, ele é mostrado ao usuário.

2.1.1.4 Bean Gerenciado

Um bean gerenciado, como dito anteriormente, é uma classe JavaBeans tradicional registrada no JSF que é gerenciada pelo framework e serve como modelo para o componente da UI [18]. Ele é responsável pelas seguintes funções:

- Sincronizar os dados no lado do servidor com os componentes registrados na árvore de componentes;
- Processar a lógica do negócio;
- Controlar a navegação entre as páginas.

O escopo padrão de um bean gerenciado é o @RequestScoped [18]. A tabela 3 mostra um breve resumo sobre cada escopo disponível [37]:

Tabela 3 - Escopo no JSF

Escopo	Descrição
@RequestScoped	O bean possui o tempo de vida igual ao de uma requisição/resposta HTTP.
@NoneScoped	O tempo de vida do bean é igual ao de uma avaliação da EL[1].
@ViewScoped	O tempo de vida do bean é igual ao tempo que usuário está interagindo com a mesma página do JSF.
@SessionScoped	O tempo de vida do bean é igual ao tempo da sessão HTTP.
@ApplicationScoped	O tempo de vida do bean é igual ao da própria aplicação.

[1] *Expression Language*

@CustomScoped	O tempo de vida do bean é determinado de acordo com a conveniência do programador.

2.1.1.5 Implementações do JSF

Existem duas grandes implementações do JSF: Oracle Mojarra e Apache MyFaces. Embora não haja muita diferença entre as duas versões já que ambas têm que aderir a especificação da API do JSF [12], o grau de documentação, manutibilidade, suporte ao usuário varia conforme a empresa responsável.

Entretanto observando o critério de performance foi percebido [37] que a implementação do MyFaces possui um melhor desempenho conforme a quantidade de componentes cresce como demonstra a figura 4.

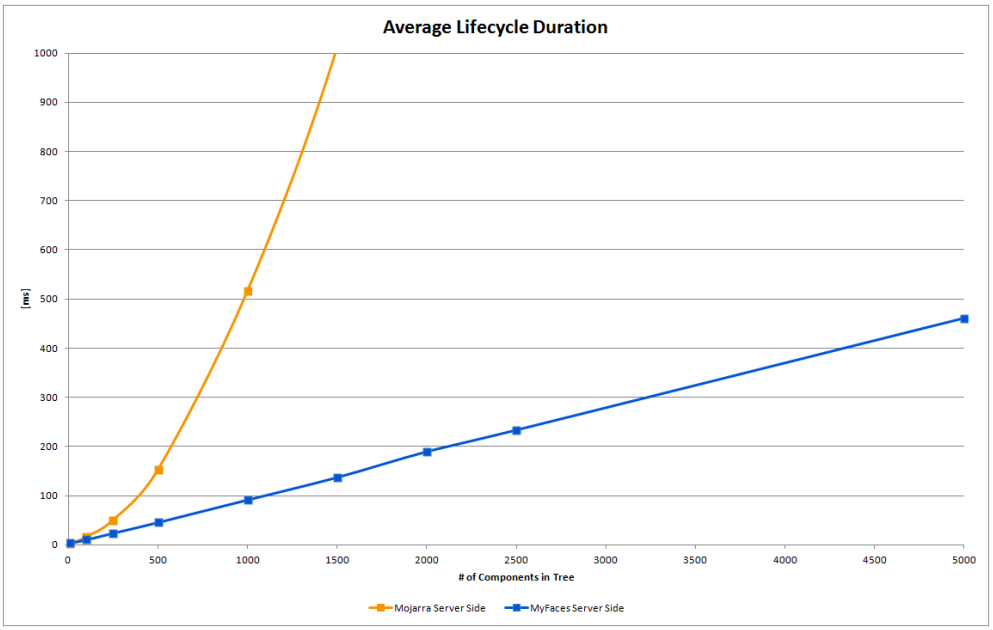


Figura 4 - Tempo de Resposta do Ciclo de Vida

2.2.2 Hibernate

Antes de falar sobre o Hibernate propriamente, é preciso entender como os dados estão representados em dois mundos diferentes em se tratando de software: um deles é o mundo Java onde só existem objetos e o outro é o mundo dos bancos de dados relacionais onde os dados são o mais importante [26]. O processo de mapear objetos Java para o mundo relacional é conhecido como mapeamento objeto-relacional (do inglês, *object-relational mapping*). Sendo assim, o Hibernate é uma implementação ORM open source [27].

Portanto sua função é auxiliar na forma como a aplicação faz a persistência dos dados [28].

2.2.2.1 ORM

Um dos maiores problemas entre o modelo de objeto e o relacional é o problema da impedância. Essa impedância se refere à diferença existente esses modelos. E essa diferença tornam o armazenamento e a recuperação de dados mais complicados [29].

As ferramentas ORM têm como objetivo reduzir a impedância nos sistemas orientados a objetos fazendo com que o programador não tenha que se preocupar com os comandos de linguagem SQL devido ao fato da aplicação ORM fazer todo o trabalho de persistência [29].

2.2.2.2 JPA

A Java Persistence API (JPA) é um framework leve baseado em POJO (do inglês, *Plain Old Java Object,* ou seja, Tradicional Objeto Java) para persistência Java [30]. Ou seja, transformando as próprias classes Java em entidades é possível inseri-las no banco de dados.

A figura 4 mostra um exemplo da classe Employee sendo transformada numa entidade persistente através da anotação @Entity.

```java
package br.cin.ufpe.healthwatcher.model.employee;

import java.io.Serializable;

@Entity
public class Employee implements Serializable {

    @Column(length = 100)
    private String name;

    @Id
    @Column(length = 100)
    private String login;

    @Column(length = 60)
    private String password;

    @Column(columnDefinition="boolean default true", nullable=false)
    private boolean enable;
```

Figura 5 - Entidade Employee

Essa entidade é transformada numa tabela através de uma API específica chamada EntityManager. E essa API é configurada para ser capaz de persistir e gerenciar tipos específicos de objetos, ler e escrever dados em vários tipos de banco de dados e é implementado de acordo com provedor de persistência especificado. Oracle, MySQL, PostgreSQL são exemplos de empresas que fornecem provedores para os seus SGBD's (Sistema Gerenciador de Banco de Dados). Nesse trabalho foi utilizado o SGBD PostgreSQL pelo fato dele ser é um poderoso sistema banco de dados objeto relacional de código aberto [31]. A figura 5 mostra a estrutura da tabela Employee gerada pela API.

	login [PK] character varying(100)	enable boolean	name character varying(100)	password character varying(60)
1	admin	TRUE	Administrator	$2a$10$EblZqNptyYvcLm/VwDCVAuBjzZOI7khzdyGPBr08PpIi0na624b8.
2	hemr	TRUE	Henrique	$2a$10$uNP.x/.oGn/J851GQmXNV.FmUS0K1QsG2Bq9C8OX44RwrmuIXO6tq
*				

Figura 6 - Tabela Employee no PostgreSQL

Embora o mapeamento objeto relacional seja o componente principal da API, ela também oferece soluções para desafios arquiteturais de integrar a persistência de forma escalável para as aplicações empresariais. Esse conjunto de API's se encontra resumido na tabela 4.

Tabela 4 - Resumo das API's do JPA

Objeto	API	Descrição
Persistence	Persistence	Classe usada para obter a instância
Entity Manager	EntityManagerFactory	Objeto utilizado para obter a
Entity Manager	EntityManager	API principal utilizada para realizar

2.2.2.3 JDBC

A API JDBC (do inglês, *Java Database Connectivity*) é instrumento padrão para acessar os dados que estão guardados em bancos de dados relacionais [32]. Com uso dessa api é possível realizar três atividades que são rotineiras no processo de desenvolvimento:

1. Conexão com a fonte de dados, tradicionalmente um banco de dados;
2. Enviar consultas e instruções de atualização para o banco de dados;
3. Recuperar e processar os resultados recebidos das consultas realizadas.

O Hibernate encapsula as funções realizadas por API para realizar o seu papel que é considerado de mais alto nível por ser capaz de acrescentar novas funcionalidades (como o mapeamento objeto-relacional, por exemplo). Na próxima seção, iremos detalhar mais sobre o papel do Hibernate.

2.2.2.4 Papel do Hibernate

Após conhecer mais sobre a API do JPA e JDBC, podemos ter uma melhor noção sobre o papel do Hibernate numa aplicação web. A figura 6 resume o relacionamento entre as entidades persistentes, JDBC e Hibernate.

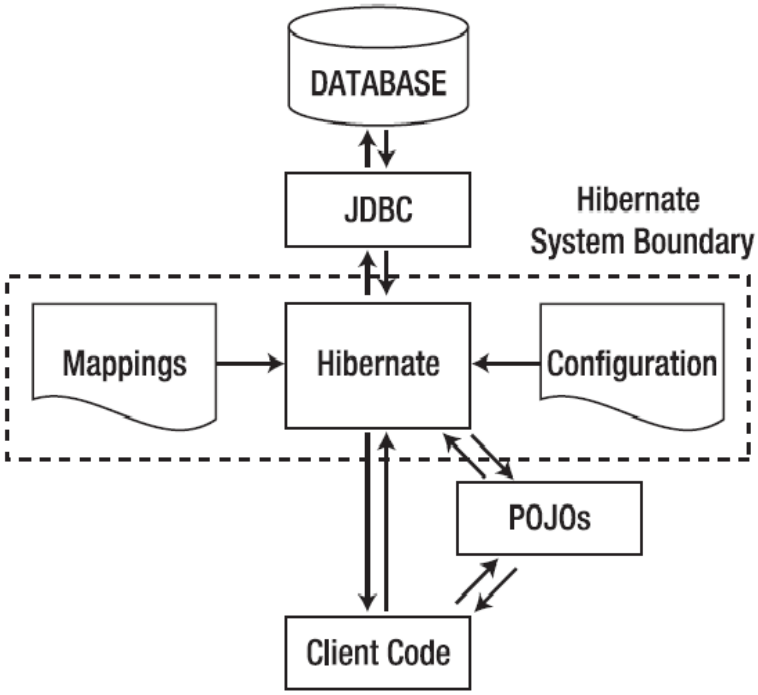


Figura 7 - Papel do Hibernate

Como podemos observar, o Hibernate, como uma solução de persistência, provê um gerenciamento de recursos de forma mais limpa, pois não é preciso se preocupar com as conexões com banco de dados, nem ter blocos de tratamento gigantescos [33]. Além disso, ele constrói o esquema do banco de dados, pode mapear um objeto para múltiplas tabelas, lidar com os

relacionamentos existentes entre objetos, dentre outras coisas. Por essas razões, fica claro o motivo pelo esse framework é tão utilizado. Cerca de 67,5% das empresas utilizam essa solução [34].

2.3 Programação orientada a aspectos

Quando se desenha um software, o processo de design mais comum é dividir o problema que se quer resolver em problemas menores. As linguagens de programação possuem mecanismos que permitem ao programador definir abstrações do sistema em subunidades (ou módulos) e compor essas abstrações de várias maneiras distintas para produzir o sistema [40]. Entretanto existe um problema chamado de entrelaçamento (do inglês, *tangling*) que torna a manutenção do software algo muito custoso, pois conforme o software cresce fica mais difícil separar os interesses (do inglês, *concerns*). Isso juntamente com o espalhamento (do inglês, *scattering*) são dois fatores chaves para deterioração da manutibilidade dos sistemas [44].

A programação orientada a aspectos (POA) nasceu fruto de uma pesquisa no Centro de Pesquisa em Palo Alto durante a década de 80 e 90 [40] e tem como objetivo primário a separação dos interesses [47], em especial, os interesses entrecortantes (do inglês, *cross-cutting concerns*) como um passo adiante para modularização [41]. A figura 8 mostra a estrutura típica de um sistema onde há uma composição de vários interesses.

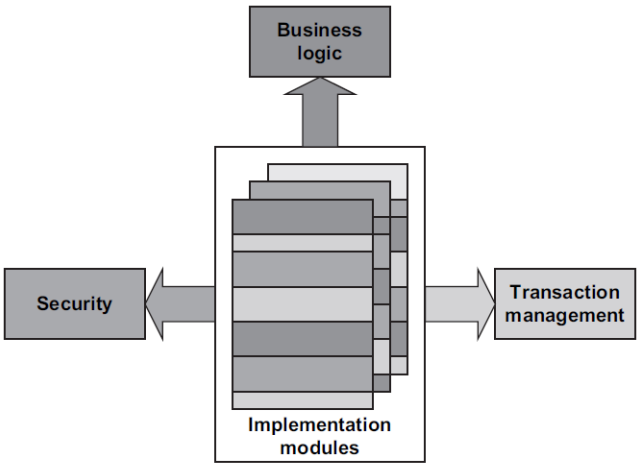


Figura 8 - Composição de um sistema

Embora haja a separação conceitual entre múltiplos interesses em tempo de modelagem, a implementação

Desde então, surgiram várias linguagens de programação que incorporaram as técnicas descritas na programação orientada a aspectos seja através de extensões como: AspectC++ (C++), AspectJ (Java), Aspect# (C#), Aquarium (Ruby) seja através mecanismos dinâmicos como Python. Nesse trabalho foi escolhido o AspectJ que será detalhado na seção 2.3.4. A seguir, será abordado dois problemas corriqueiros quando tratamos sobre interesses entrecortantes: o entrelaçamento e o espalhamento.

2.3.1 Entrelaçamento

O entrelaçamento ocorre quando um módulo é implementado para lidar com múltiplos interesses simultaneamente [2]. Isso faz com que vários elementos que representam interesses diferentes estejam presentes na implementação, o que resulta no entrelaçamento. A figura 9 mostra uma representação gráfica desse problema.

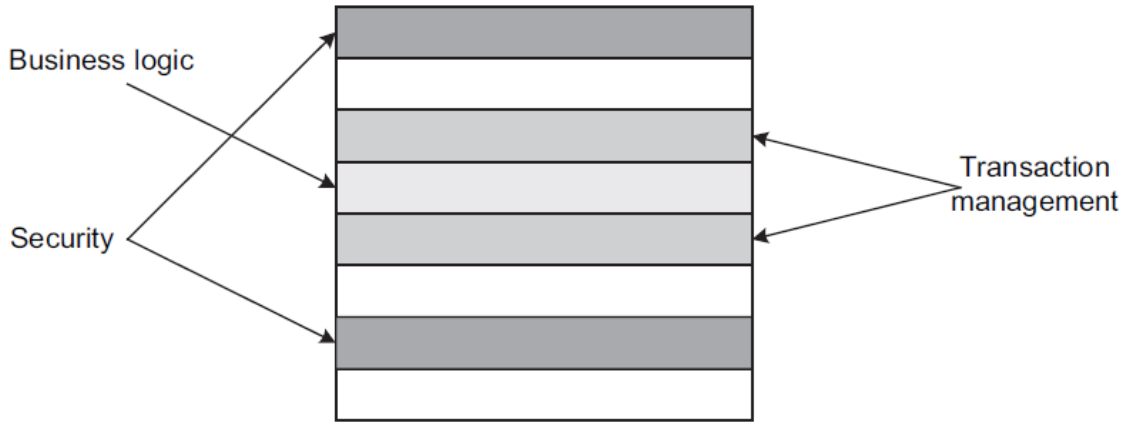


Figura 9 - Entrelaçamento

Outra forma de entender esse problema é entender o interesse como algo multidimensional enquanto a implementação é algo unidimensional. Quando a representação do interesse é mapeada pela implementação, essas dimensões extras já inseridas numa única dimensão. A figura 10 mostra como isso ocorre.

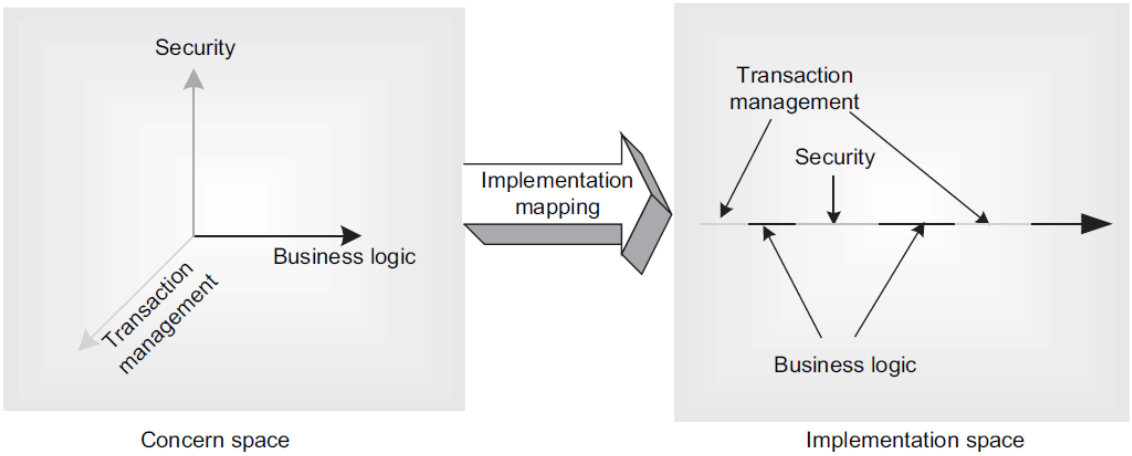


Figura 10 - Interesses numa única dimensão

Outro problema do entrelaçamento é que torna mais difícil o reuso da função primária do módulo em outro cenário já que o código estará entrelaçado com outras funcionalidades secundárias [46].

2.3.2 Espalhamento

O espalhamento ocorre quando uma única funcionalidade é implementada em vários módulos [2]. Isso decorre pelo fato dos interesses entrecortantes estarem espalhados em inúmeros módulos. A figura 11 mostra um exemplo desse fenômeno.

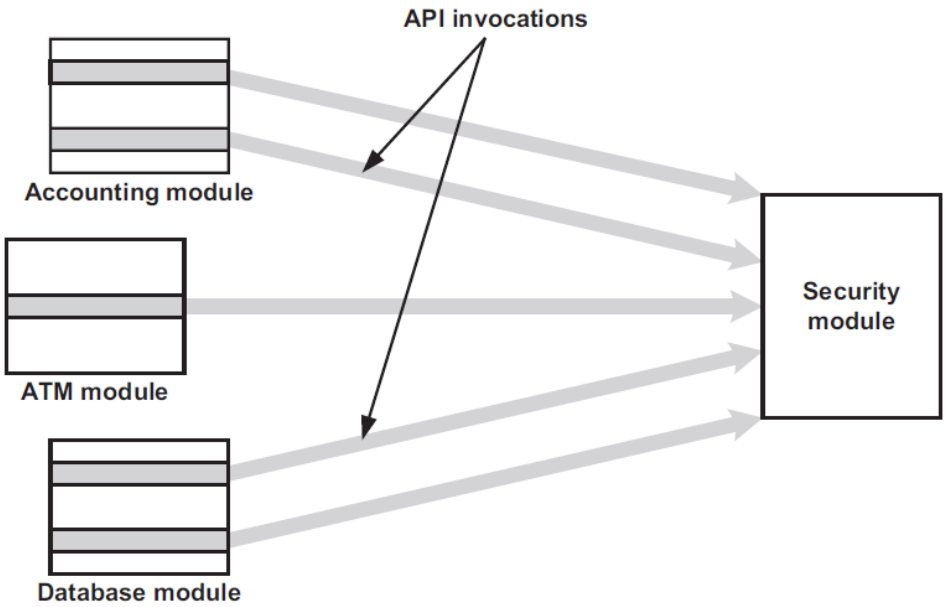


Figura 11 - Espalhamento

Como demonstrado na figura 11, existem várias chamadas à API do módulo de segurança espalhadas em vários módulos do sistema o que faz com que haja um acoplamento entre esses módulos.

Sendo assim, podemos perceber que tanto entrelaçamento quanto espalhamento impactam diretamente no design e no desenvolvimento de software em várias formas, causando problemas como: pouca rastreabilidade, menor produtividade, menor reuso de código, pouca qualidade e difícil evolução [2]. Na próxima seção, será detalhado sobre o uso do AspectJ e como ele funciona para que sobre

2.3.3 AspectJ

O AspectJ é uma extensão orientada a aspectos feita para linguagem de programação Java [2, 45]. Ela possibilita a modularização de forma limpa dos interesses entrecortantes, tais como, tratamento de erros, sincronização, comportamento sensível a contexto, otimizações de performance, monitoramento e logging, suporte à depuração e protocolos multi-objetos. Como toda implementação de POA, o AspectJ consiste em duas partes: a primeira parte é especificação da linguagem que cuida da gramática e da semântica da linguagem e a segunda parte é implementação da linguagem que verifica a aderência do código a especificação da linguagem e traduz o código numa forma executável [2].

Para que o AspectJ saiba exatamente o local no código no qual ele deverá entrecortar algum interesse é preciso que existam marcadores especiais que permitam isso. O compilador do AspectJ pode encontrar os locais no código fonte onde algumas operações pré-definidas são executadas: chamadas de métodos, atribuição de valores, etc. Essas operações pré-definidas ou pontos bem determinados na execução do programa são chamados de join points [48]. Na próxima seção, será detalhado mais sobre esses join points.

2.3.3.1 Join Points

Um join point é uma localização bem definida dentro de um código onde o interesse irá entrecortar a aplicação [47]. A tabela 5 mostra os 11 tipos de join points disponíveis na linguagem.

Tabela 5 - Join Points

Join Points
Chamada de método
Execução de método
Chamada do construtor
Execução do inicializador
Execução do construtor
Execução do inicializador estático
Pré-inicialização do objeto
Inicialização do objeto
Referência do atributo
Atribuição do atributo
Execução do tratamento de execução

A linguagem AspectJ define um construtor sintático para detectar esses join points e é chamado de designador de pointcut [48] (do inglês, *pointcut designator*).

2.3.3.2 Pointcuts

Um pointcut é um construto do programa que seleciona e coleta o contexto dos joint points [2]. Ele também pode ser definido como um elemento do programa que seleciona os join points e expõe os dados do contexto de execução desses join points [49]. Eles são usados primariamente pelos advices e podem ser compostos por operadores booleanos para construir outros pointcuts. De uma forma geral, a declaração de um pointcut deve obedecer a seguinte sintaxe:

Pointcut Nome(argumentos): corpo;

A figura 12 mostra um exemplo de pointcut.

```
pointcut transactionalMethods(): execution(* HealthWatcherFacade.*(..)) &&
                                 ! execution(* HealthWatcherFacade.*Record(..))  &&
                                 ! execution(static * *.*(..));
```

Figura 12 - Pointcut

2.3.3.3 Advice

O Advice é o código executado pelo join point selecionado pelo pointcut [2, 48]. Na maior parte das especificações POA, o código do advice será executado em três lugares diferentes quando um join point é correspondido:

antes (do inglês, *before*), ao redor (do inglês, *around*) e depois (do inglês, *after*) [47]. A figura 13 mostra um exemplo de advice que será executado antes do pointcut chamado transactionalMethods.

```
before(): transactionalMethods() {
    getPm().beginTransaction();
}
```

Figura 13 – Advice

2.3.4 AspectJ Development Tools (AJDT)

O projeto AspectJ Development Tools é uma ferramenta de suporte integrada a plataforma Eclipse para o desenvolvimento orientado a aspectos com AspectJ [50]. Esse trabalho utilizou o Eclipse como plataforma de desenvolvimento, portanto graças ao AJDT foi possível dar continuidade ao processo de conversão da versão original do Health Watcher para a versão atual aderente a Plataform JEE 6, e a posteriori, a versão AOP.

3. Health Watcher

O Health Watcher (HW) é um sistema de informação baseado na plataforma web [51]. Seu objetivo é coletar e controlar as reclamações e notificações e prover informações importantes para as pessoas sobre o sistema de saúde [52] da cidade de Recife. A versão original desse sistema foi desenvolvida pelo grupo de pesquisa de Produtividade de Software da Universidade Federal de Pernambuco em duas versões: OO e AO [53].

Nesse trabalho, esse sistema foi adaptado para a plataforma JEE atual. Tanto a versão OO original quanto a versão OO nova foram implementadas usando a linguagem de programação Java. Além disso, foi desenvolvida uma versão AO (do inglês, *Aspect-Oriented*) implementada com AspectJ. Nas próximas seções, iremos detalhar o design arquitetural do sistema em cada uma dessas versões.

Como ambiente de deploy da aplicação foi utilizado o container Apache Tomcat. Ele é uma implementação em código aberto [55] de uma série de tecnologias que permitem que uma aplicação JEE funcione.

3.1 Design Orientado a Objetos na versão original

O padrão arquitetural Layer [54] foi utilizado para representar a organização do sistema de classes da aplicação original como demonstra a figura 14. Essas classes foram agrupadas em quatro camadas: GUI (do inglês, *Graphical User Layer*), Distribuição (do inglês, *Distribuition*), Negócio (do inglês, *Business*) e Dados (do inglês, *Data*). É importante observar que foi representado apenas uma parte das classes do sistema.

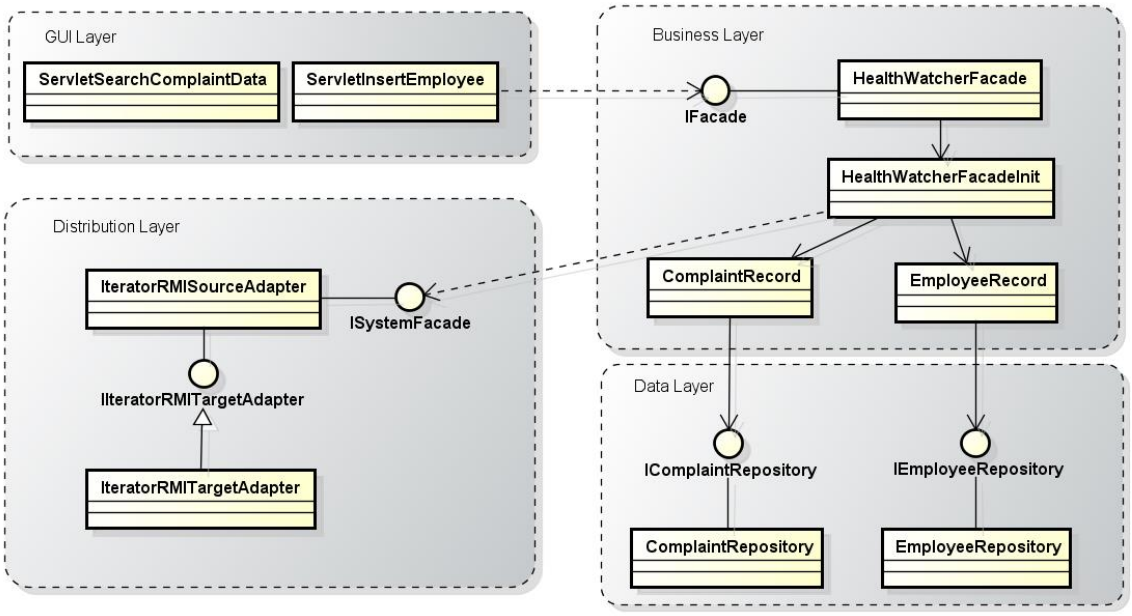


Figura 14 - Arquitetura HW versão original OO

A camada de GUI implementa a interface web para o usuário. Isso é feito a partir da implementação da API Java Servlet. A camada de Distribuição é responsável por tornar os serviços da aplicação distribuídos através da camada de Negócio. Isso é implementado através da tecnologia RMI. A camada de Negócio possui as classes que implementam as regras de negócio da aplicação. Por fim, a camada de Dados permite que os dados sejam persistidos no banco de dados através da API JDBC.

3.2 Design Orientado a Objetos na nova versão

A distribuição de camadas na nova versão do sistema foi radicalmente alterada, pois com uso do Tomcat não foi necessário utilizar implementar os serviços de distribuição. Além disso, a camada de GUI foi substituída pelo uso de classes em formato xhtml como mostra a figura 15.

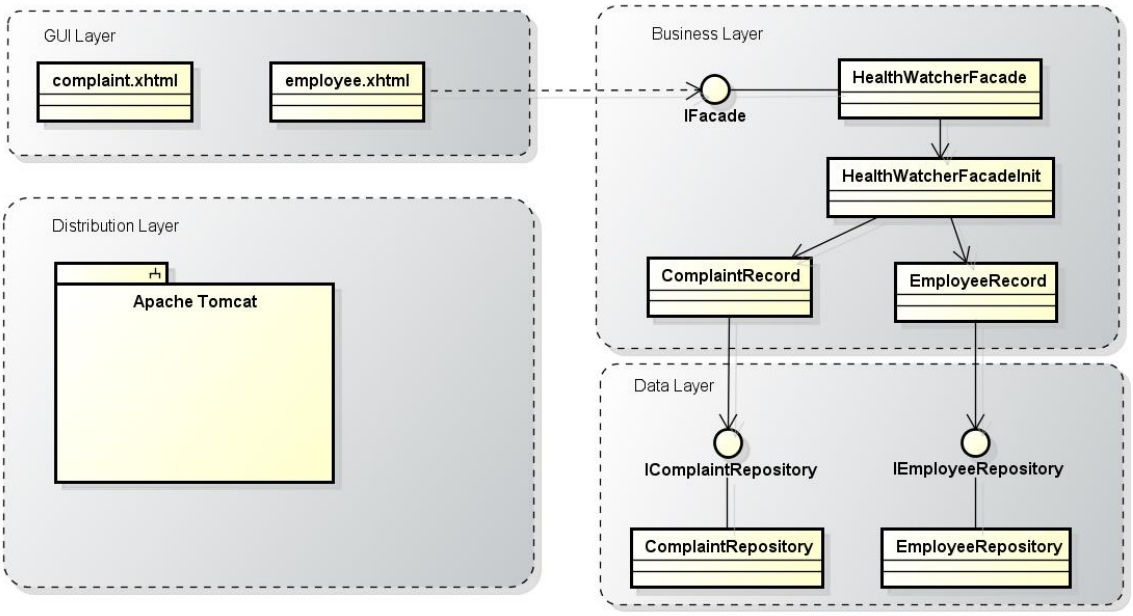


Figura 15 - Arquitetura HW Nova Versão OO

3.3 Design Orientado a Aspectos na nova versão

A arquitetura do sistema não sofreu alteração da nova versão OO. Mas conforme será demonstrado na próxima seção, vários interesses que estavam espalhados nas várias classes foram concentrados graças ao uso dos aspectos.

4. Analisando o AspectJ nas versões do sistema Health Watcher

Nesse estudo foi utilizada uma suíte de métricas para separação de interesses (do inglês, *Separation of Concerns*) e tamanho como critérios de avaliação das implementações do Health Watcher nas versões OO e AO. Essas métricas já foram utilizadas em vários estudos de caso [53, 54] e auxiliam bastante no processo de validação dos benefícios do uso da programação orientada a aspectos. A tabela 6 define brevemente cada uma das métricas e associa com seu atributo de software relevante.

Tabela 6 – Métricas

Atributos	Métricas	Definição
Separação de interesses	Concern Diffusion over Components [2](CDC)	Contabiliza o número de classes e aspectos cujo propósito principal é contribuir para implementação de um interesse e o número de outras classes ou aspectos que acessam a elas.
	Concern Diffusion over Operations[3] (CDO)	Contabiliza o número de métodos e advices cujo propósito principal para implementação de um interesse e o número de outros métodos ou advices que acessam a elas.
	Concern Diffusion over LOC[4] (CDLOC)	Contabiliza o número de pontos de transição para cada interesse através das linhas de código. Pontos de transição são pontos no código onde há uma "troca de interesse".
Tamanho	Lines of Code[5] (LOC)	Contabiliza as linhas de código.
	Number of Attributes[6] (NOA)	Contabiliza o número de atributos de cada classe ou aspecto.
	Weighted Operations per Component[7] (WOC)	Contabiliza o número de métodos ou advices de cada classe ou aspecto e número dos seus parâmetros.

[2] Em tradução livre, Difusão de Interesses nos Componentes.
[3] Em tradução livre, Difusão de Interesses nas Operações.
[4] Em tradução livre, Difusão de Interesses por Linhas de Código
[5] Em tradução livre, Linhas de Código
[6] Em tradução livre, Número de atributos
[7] Em tradução livre, Operações ponderadas por componente

4.1 Procedimentos para avaliação

Como forma de avaliação, foram utilizadas como parâmetros de comparação quatro versões do Health Watcher: (i) Base OO (versão OO original), (ii) Base AO (versão AO original), (iii) New OO (nova versão OO) e (iv) New AO (nova versão AO). Em todas as versões do sistema foram implementados um total de 8 casos de uso como demostrado na tabela 7.

Todas as versões do sistema se encontram disponíveis para download no GitHub através dos seguintes endereços:

- Base OO: https://github.com/fagnerffcs/HealthwatcherOO_01_Base;
- Base AO: https://github.com/fagnerffcs/HealthwatcherAO_01_Base;
- New OO: https://github.com/fagnerffcs/HealthwatcherOO_01_New;
- New AO: https://github.com/fagnerffcs/HealthwatcherAO_01_New.

Tabela 7 - Casos de Uso

Casos de Uso
Inserir, Atualizar e Buscar um Funcionário
Inserir, Atualizar e Buscar uma Reclamação
Atualizar e Procurar uma lista de Unidades de Saúde
Buscar Especialidades por Unidade de Saúde
Buscar Unidades de Saúde por Especialidade
Buscar Lista de Especialidades
Buscar os tipos de doenças
Buscar a lista de tipos de doenças

Com base nessas quatro versões do sistema Health Watcher e a utilização das métrica descritas nessa seção foi possível fazer a comparação das versões e inferir os resultados descritos na próxima seção.

4.2 Comparação de resultados

A aplicação das métricas SoC[8] foi um instrumento essencial para quantificar o quão efetiva foi a separação dos interesses de persistência.

[8] Separation of Concerns (SoC)

Através dessa parametrização, foram identificadas as métricas descritas na tabela 8.

Tabela 8 - Valores coletados para as métricas por Separação de Interesses

Interesse		**Persistência**		
Métrica		**CDC**	**CDO**	**CDLOC**
HW	**Base OO**	56	286	353
	Base AO	35	206	17
	New OO	38	174	153
	New AO	15	123	91

Como pode ser observado a implementação da versão AO mostrou diferenças significativas no que se refere ao CDC, CDO e CDLOC. Isso já era esperado devido as características inerentes ao uso da POA que permite com que os interesses que estavam difusos nas classes fiquem concentrados nos aspectos. O gráfico da figura 16 representa esse processo de redução de forma visual.

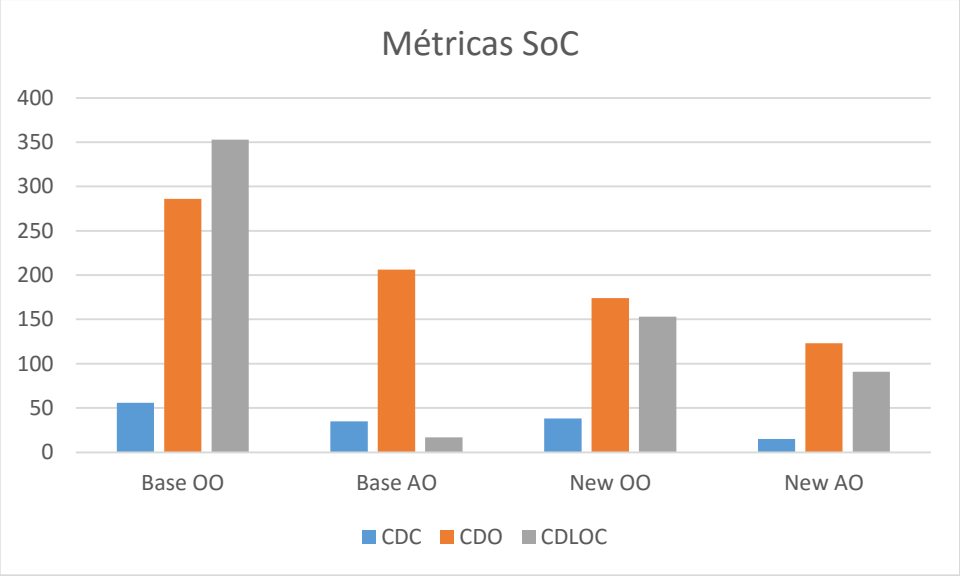


Figura 16 – Métricas SoC

Outro fato interessante a ser observado é que houve uma diminuição nas métricas nas versões mais recentes do Health Watcher. Isso se dá devido a melhor separação entre a camada de negócio e de visão que JSF permite. Enquanto na versão anterior existe um Servlet para cada operação CRUD

(Create-Read-Update-Delete), na nova versão um bean gerenciado pode cuidar de todas essas operações.

Um detalhe importante a ser destacado é que com a utilização das anotações para mapear os atributos do objeto para modelo relacional, isso fez com que esses atributos recebessem mais de um poincut. Isso será melhor detalhado na seção 4.2.2 adiante.

Com uso da ferramenta CodePro AnalytiX [56], foi possível coletar as métricas relativas ao tamanho do sistema como pode ser visto na tabela 9.

Tabela 9 - Valores coletados para as métricas por Tamanho

Métrica		**Tamanho**		
		LOC	**NOA**	**WOC**
HW	**Base OO**	6.114	189	1.154
	Base AO	5.759	188	1.132
	New OO	4.603	231	839
	New AO	4.074	238	831

Podemos perceber que com a utilização do AspectJ, houve uma diminuição na quantidade de linhas de código na versão AO do sistema em relação à OO. Isso ocorre devido a isolação e a localização dos interesses que a programação orientada a aspectos permite.

Entretanto existem alguns interesses no contexto da persistência onde o uso da POA não permitiu um maior reuso dos interesses: Named Query e o mapeamento dos relacionamentos entre as entidades através do JPA.

4.2.1 Named Query

Uma *named query* é uma consulta pré-definida que o programador cria e associa a uma entidade [57] gerenciada pelo contexto de persistência. Assim, não é necessário que ele escreva a mesma consulta toda vez que for preciso utilizá-la sendo necessário apenas informar ao Entity Manager o nome da consulta e os parâmetros, quando houver. A figura 17 mostra um exemplo de named query onde a consulta retorna todos os registros do tipo AnimalComplaint do banco de dados.

```java
@SuppressWarnings("unchecked")
public List<AnimalComplaint> listAnimalComplaints(){
    EntityManager em = new JPAUtil().getEntityManager();
    List<AnimalComplaint> lista = em.createNamedQuery("allAnimalComplaints").getResultList();
    em.close();
    return lista;
}
```

Figura 17 - Exemplo Named Query

Na versão OO do sistema, essas consultas foram definidas no início das classes de domínio, ou seja, nas classes que possuem a anotação *@Entity*. A figura 18 mostra o exemplo no qual são definidas essas consultas na classe Animal Complaint. Importante observar que as consultas estão escritas na linguagem JPQL[9] e são traduzidas para o formato SQL pela API do Hibernate. Quando existem mais de uma consulta @NamedQuery definida numa mesma classe é preciso utilizar a anotação @NamedQueries.

```java
@Entity
@NamedQueries({
    @NamedQuery(name="animalComplaintByCode", query="SELECT a FROM AnimalComplaint a WHERE a.codigo = :code"),
    @NamedQuery(name="allAnimalComplaints", query="SELECT a FROM AnimalComplaint a"),
    @NamedQuery(name="animalComplaintsBySituation", query="SELECT a FROM AnimalComplaint a WHERE a.situacao = :situacao"),
})
public class AnimalComplaint extends Complaint implements Serializable {
```

Figura 18 - Named Query HW OO

Já na versão AO do sistema, as consultas que estavam definidas e espalhadas naquelas classes de domínio logo acima do nome dessas classes foram localizadas num único arquivo que representa o interesse de consultas JPA graças ao uso da POA. Sendo assim, esse aspecto foi chamado de HWQueries e ele é o responsável por agrupar todas essas consultas como pode ser percebido pela figura 19.

[9] *Java Persistence Query Language*, ou seja, a linguagem de persistência utilizada pelo Java.

```
public aspect HWQueries {

    //employee package
    declare @type : br.cin.ufpe.healthwatcher.model.employee.Employee :
    @NamedQueries({
        @NamedQuery(name="todosEmployees", query="SELECT e FROM Employee e"),
        @NamedQuery(name="employeePorLoginSenha", query="SELECT e FROM Employee e WHERE e.login = :login AND e.password = :password"),
    });

    //complaint package
    declare @type : br.cin.ufpe.healthwatcher.model.complaint.AnimalComplaint :
    @NamedQueries({
        @NamedQuery(name="animalComplaintByCode", query="SELECT a FROM AnimalComplaint a WHERE a.code = :code"),
        @NamedQuery(name="allAnimalComplaints", query="SELECT a FROM AnimalComplaint a"),
        @NamedQuery(name="animalComplaintsBySituation", query="SELECT a FROM AnimalComplaint a WHERE a.situacao = :situacao"),
    });

    declare @type : br.cin.ufpe.healthwatcher.model.complaint.FoodComplaint :
    @NamedQueries({
        @NamedQuery(name="foodComplaintByCode", query="SELECT f FROM FoodComplaint f WHERE f.code = :code"),
        @NamedQuery(name="allFoodComplaints", query="SELECT f FROM FoodComplaint f"),
        @NamedQuery(name="allFoodComplaintsBySituation", query="SELECT f FROM FoodComplaint f WHERE f.situacao = :situacao"),
    });

    declare @type : br.cin.ufpe.healthwatcher.model.complaint.SpecialComplaint :
        @NamedQueries({
            @NamedQuery(name="medicalSpecialtyByCode", query="SELECT m FROM MedicalSpecialty m WHERE m.code = :code"),
            @NamedQuery(name="listAllMedicalSpecialties", query="SELECT m FROM MedicalSpecialty m")
        });

    //healthguide package
    declare @type : br.cin.ufpe.healthwatcher.model.healthguide.HealthUnit :
    @NamedQueries({
        @NamedQuery(name="healthUnitByName", query="SELECT h FROM HealthUnit h WHERE h.code = :code"),
        @NamedQuery(name="allHealthUnits", query="SELECT h FROM HealthUnit h"),
        @NamedQuery(name="healthUnitsBySpecialty", query="SELECT h FROM HealthUnit h inner join h.specialties specialties WHERE specialties.code IN :code")
    });
```

Figura 19 - Named Query HW AO

Assim podemos perceber que não houve redução no número de linhas entre as duas versões do sistema, existindo ainda sim, o benefício da localização de interesse.

4.2.2 Mapeamento Objeto-Relacional

Como descrito na seção sobre ORM, através do mapeamento objeto relacional descrito pela API do JPA é responsável por automatizar de modo transparente a persistência dos objetos Java para o banco de dados relacional.

Durante o processo de mapeamento das associações existentes entre as classes persistentes é muito comum que sejam utilizadas anotações como: @OneToOne, @OneToMany, @ManyToOne e @ManyToMany, o que torna essas anotações elegíveis para separação através da POA. No contexto do Health Watcher, por exemplo, existe o mapeamento do relacionamento da entidade Address com a entidade AnimalComplaint, na qual vemos que dentro da anotação *@JoinColumn* existe um atributo chamado name que determina o nome da chave estrangeira em Animal Complaint como demonstra a figura 19.

```java
public class AnimalComplaint extends Complaint implements Serializable {
    private static final long serialVersionUID = 1L;

    private Short animalQuantity;

    @Temporal(TemporalType.DATE)
    private Date inconvenienceDate;

    @Column(length = 100)
    private String animal;

    @ManyToOne(cascade=CascadeType.ALL)
    @JoinColumn(name = "code")
    private Address occurenceLocalAddress;
```

Figura 20 - Anotação @JoinColumn HW OO

Como pode ser notado, o atributo *occurenceLocalAddress* possui duas anotações que serão refletidas em mais de um poincut como descrito anteriormente. Afinal através do AspectJ, só é possível inserir uma anotação por vez em cada atributo. Embora ainda sim seja possível inserir essa anotação @JoinColumn em todas as classes que possuem como atributo um campo do tipo Address como mostra a figura 21.

```java
//@JoinColumn mapping
declare @field : (br.cin.ufpe.healthwatcher.model.address.Address
                  br.cin.ufpe.healthwatcher.model.complaint.AnimalComplaint.occurenceLocalAddress) ||
                 (br.cin.ufpe.healthwatcher.model.address.Address
                  br.cin.ufpe.healthwatcher.model.complaint.SpecialComplaint.enderecoOcorrencia) ||
                 (br.cin.ufpe.healthwatcher.model.address.Address
                  br.cin.ufpe.healthwatcher.model.complaint.FoodComplaint.enderecoDoente):
                  @JoinColumn(name = "addressCode");
```

Figura 21 - @JoinColumn HW AO

Todavia quando é necessário que definir algum tipo de particularidade no mapeamento de uma associação é preciso separá-la das demais, o que dificulta o reuso como pode ser percebido no destaque da figura 22.

```java
//@JoinColumn mapping
declare @field : (br.cin.ufpe.healthwatcher.model.address.Address
                  br.cin.ufpe.healthwatcher.model.complaint.AnimalComplaint.occurenceLocalAddress) ||
                 (br.cin.ufpe.healthwatcher.model.address.Address
                  br.cin.ufpe.healthwatcher.model.complaint.SpecialComplaint.enderecoOcorrencia) ||
                 (br.cin.ufpe.healthwatcher.model.address.Address
                  br.cin.ufpe.healthwatcher.model.complaint.FoodComplaint.enderecoDoente):
                  @JoinColumn(name = "addressCode");

declare @field : (br.cin.ufpe.healthwatcher.model.address.Address
                  br.cin.ufpe.healthwatcher.model.complaint.Complaint.enderecoSolicitante):
@JoinColumn(name = "enderecoSolicitante_code");
```

Figura 22 - Problema no @JoinColumn HW AO

Nesse caso em questão foi possível retirar a anotação @JoinColumn das classes AnimalComplaint, FoodComplaint e SpecialComplaint, mas dentro da superclasse Complaint não foi possível, pois senão haveria um conflito de atributos com o mesmo nome no banco de dados.

Outro problema similar a esse ocorre quando utilizamos a anotação @GeneratedValue. Essa anotação é utilizada para informar ao banco qual será a estratégia para geração dos valores da chave primária [58]. Uma dessas estratégia é o uso de *sequences* para gerar esses valores através das próprias sequences no banco de dados. A figura 22 mostra um exemplo desse tipo de estratégia.

```
@MappedSuperclass
public abstract class Complaint implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="SEQUENCE_COMPLAINT")
    private int codigo;
```

Figura 23 - Sequence Generator

Contudo é possível que, na implementação da estratégia de geração de chaves, se queira utilizar uma sequence para cada entidade/tabela no banco de dados. Com isso, não seria possível modularizar esse aspecto de modo otimizado.

5. Conclusão e Trabalhos Futuros

Esse trabalho foi dividido essencialmente em três partes: a adaptação da versão base do Health Watcher para o cenário desenvolvimento web atual; a evolução dessa nova versão do sistema com a introdução da programação orientada a aspectos através do AspectJ; e a comparação das versões destacando os benefícios e desvantagens de uma em relação a outra.

Como podemos perceber nesse trabalho, a utilização da programação orientada a aspectos ainda possibilita uma série de benefícios para os programadores como: a modularização e localização dos interesses, maior reuso de código, manutenção e correção de bugs de forma centralizada, dentre outras coisas.

Entretanto podemos perceber que com a expansão do uso das anotações na plataforma JEE, boa parte dessa modularização fica prejudicada devido as possibilidades de customização dessas anotações através das propriedades existentes em cada anotação.

Este trabalho foi concebido para validação do sistema Health Watcher considerando todas as suas áreas de interesse. Dessa forma, algumas limitações desse trabalho são conhecidas e futuras ações são esperadas. Dentre elas, podemos destacar as seguintes:

- Avaliação das demais áreas de interesse: validação, distribuição, tratamento de exceção e logging, não se concentrando apenas na parte de persistência. Assim, é possível perceber até que ponto a utilização da programação orientada a aspectos contribuiu na nova versão do sistema;
- Utilização de ferramentas de análise de código apropriadas para avaliação dos aspectos como: AJATO e AOP Metrics. Assim, é possível coletar as informações de cada classe e avaliar outras métricas como: acoplamento e coesão;
- O processo de deploy da aplicação na versão AO no Tomcat apresentou alguns problemas na persistência das entidades o que indica uma possível incompatibilidade entre o processo de persistência usando a geração automática de chaves primárias com a compilador do AspectJ.

Em suma, este trabalho contribuiu no entendimento sobre a relevância do uso do paradigma orientado a aspectos através do AspectJ nos frameworks Java atuais. Mesmo com a expansão do uso de anotações na plataforma JEE, é possível aferir benefícios com uso do AspectJ. Sendo assim, com as futuras alterações da versão AO do Health Watcher, se espera que esses benefícios e desvantagens sejam compreendidos com maior clareza pela comunidade de desenvolvimento de aplicações web com a linguagem Java.

6. Referências

[1] MATKOVIć, Predrag; TUMBAS, Pere. A Comparative Overview of the Evolution of Software Development Models. International Journal Of Industrial Engineering And Managem Ent. Online, p. 163-172. 2010. <http://www.iim.ftn.uns.ac.rs/casopis/volume1/ijiem_vol1_no4_4.pdf>. Acesso 09 set. 2010.

[2] LADDAD, Ramnivas. AspectJ in Action. 2. ed. United States Of America: Manning Publications Co, 2010.

[3] KICZALES et al. Aspect-Oriented Programming. http://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf. Acesso 22 set. 2015.

[4] BIEN, Adam. Simplicity by Design. http://www.oracle.com/au/products/database/o11java-195110.html. Acesso 10 set. 2015.

[5] CUNNINGHAM, H. Conrad; LIU, Yue; ZHANG, Cuihua. Using classic problems to teach Java framework design. http://www.sciencedirect.com/science/article/pii/S0167642305000900. Acesso 26 dez. 2015.

[6] FRANZINI, Fernando. Listagem de frameworks Java. http://imasters.com.br/linguagens/java/listagem-de-frameworks-java/?trace=1519021197&source=single. Acesso 27 dez. 2015.

[7] SCHALK, Chris. Introduction to JavaServer Faces – What is JSF? http://www.oracle.com/technetwork/topics/index-090910.html. Acesso 27 dez. 2015.

[8] The Wonderful World of JEE Plataform. JSF 2: A brief recall. https://karimdjaafar.wordpress.com/2013/01/25/jsf-2-a-brief-recall/. Acesso 27 dez. 2015.

[9] IBM.Design with JSF architecture. http://www.ibm.com/developerworks/library/wa-dsgnpatjsf/. Acesso 27 dez. 2015.

[10] ORACLE. The Java EE Tutorial. http://docs.oracle.com/javaee/6/tutorial/doc/girch.html. Acesso 27 dez. 2015.

[11] VOGEL, Lars. JSF (JavaServer Faces) – Tutorial. http://www.vogella.com/tutorials/JavaServerFaces/article.html#jsf. Acesso 27 dez. 2015.

[12] TECHUB. JSF implementations and component libraries [closed] - TecHub. http://www.tech.theplayhub.com/jsf_implementations_and_component_libraries_closed. Acesso 28 dez. 2015.

[13] GRIFIN, Neil. The Decisive Moment in the History of JSF - Blog - Liferay.com. https://www.liferay.com/web/neil.griffin/blog/-/blogs/the-decisive-moment-in-the-history-of-jsf. Acesso 28 dez. 2015.

[14] SCHOLTZ, Bauke. The BalusC Code: Disadvantages of JSF, a bit of history. http://balusc.omnifaces.org/2010/09/disadvantages-of-jsf-bit-of-history.html. Acesso 28 dez. 2015.

[15] HIGHTOWER, Rick. JSF 2.x Update from Ed Burns. http://www.infoq.com/news/2012/01/jsf-update-2.x. Acesso 29 dez. 2015.

[16] BEZEMER, C. et al. Enabling multi-tenancy: An industrial experience report. 2010 IEEE International Conference on Software Maintenance, 2010.

[17] TIJMS, Arjan. What's new in JSF 2.3? http://arjan-tijms.omnifaces.org/p/jsf-23.html. Acesso 29 dez. 2015.

[18] LAYKA, Vishal. Learn Java for Web Development. United States of America: Apress, 2014.

[19] ORACLE. The Lifecycle of a JavaServer Faces Application - The Java EE 6 Tutorial. http://docs.oracle.com/javaee/6/tutorial/doc/bnaqq.html. Acesso 2 jan. 2016.

[20] ORACLE. Java SE Desktop Technologies - Java Beans. http://www.oracle.com/technetwork/articles/javase/index-jsp-138795.html. Acesso 2 jan. 2016.

[21] FARIA, Thiago. Java EE 7 com JSF, Primefaces e CDI. 2ª ed. Minas Gerais: AlgaWorks, 2015.

[22] THE MIDDLEWARE COMPANY. History of Enterprise Java. http://java.coe.psu.ac.th/FreeOnline/TheServerSide.com/UnderstandingJ2EETV/J2EE.pdf. Acesso 2 jan. 2016.

[23] WIKIPEDIA. Java Platform, Enterprise Edition. https://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition. Acesso 2 jan. 2016.

[24] GUTENBERG, P. Java EE version history. http://www.gutenberg.us/articles/java_ee_version_history. Acesso 2 jan. 2016.

[25] WIKIPEDIA. https://en.wikipedia.org/wiki/Java_EE_version_history. Acesso 2 jan. 2016.

[26] KONDA, Madhusudhan. Just Hibernate. 2 ed. United States: Manning Publications Co, 2014.

[27] BAUER, Christian; KING, Gavin. Hibernate in Action. 2 ed. United States: Manning, 2005.

[28] HIBERNATE. What is Object/Relational Mapping? http://hibernate.org/orm/what-is-an-orm. Acesso 3 jan. 2016.

[29] PLÁCIDO, Daniel Guessi; CUNHA, Daniel Pezzi da. Impedância de Dados. http://periodicos.unesc.net/sulcomp/article/download/2004/1897. Acesso 3. Jan. 2016.

[30] KEITH, Mike; SCHINCARIOL, Merrick. Pro JPA 2. New York: Apress, 2009.

[31] POSTGRESQL. PostgreSQL: About. http://www.postgresql.org/about/. Acesso 4 jan. 2016.

[32] ORACLE. Lesson: JDBC Introduction. https://docs.oracle.com/javase/tutorial/jdbc/overview/index.html. Acesso 4 jan 2016.

[33] OTTINGER, Joseph B.; MINTER, Dave; LINWOOD, Jeff. Beginning Hibernate. 3 ed. United States: Apress, 2014.

[34] WHITE, Oliver. Java Tools and Technologies Landscape for 2014. http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/10/. Acesso 4 jan. 2016.

[35] WILLIAMS, Nicholas. Professional Java for Web Applications. United States: Wiley, 2014.

[36] MAPLE, Simon. The Ultimate Java Web Frameworks Comparison: Spring MVC, Grails, Vaadin, GWT, Wicket, Play, Struts and JSF. http://zeroturnaround.com/rebellabs/the-curious-coders-java-web-frameworks-comparison-spring-mvc-grails-vaadin-gwt-wicket-play-struts-and-jsf/. Acesso 5 jan. 2016.

[37] TUTORIALSPOINT. JSF - Managed Beans. http://www.tutorialspoint.com/jsf/jsf_managed_beans.htm. Acesso 5 jan. 2016.

[38] TECHSCOUTING THROUGH THE JAVA NEWS. JSF-Comparison: MyFaces vs. Mojarra. https://blog.oio.de/2013/04/08/jsf-comparison-myfaces-vs-mojarra/. Acesso 5 jan. 2016.

[39] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant' Anna, S. Soares, P. Borba, U. Kulesza and A. Rashid. On the impact of aspectual decompositions on design stability: An empirical study, in Proceedings of the 21st European Conference on Object-Oriented Programming, LNCS, Springer-Verlag, 2007, pp. 176–200.

[40] KICZALES et al. Aspected-Oriented Programming. https://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf. Acesso 4 de jan. 2016.

[41] KICZALES, Gregor. Aspect Oriented Programming and Modular Reasoning. http://www.ptidej.net/courses/ift6251/fall06/presentations/060920/060920%20-%20Aspect%20Oriented%20Programming%20and%20Modular%20Reasoning.pdf. Acesso 6 jan. 2016.

[42] APEL et al. Feature-Oriented Software Product Lines: Concepts and Implementation. Springer, 2013.

[43] REBÊLO, H. et al. Quantifying the effects of aspectual decompositions on Design by Contract Modularization: a maintenance study. Int. J. Soft. Eng. Knowl. Eng., v. 23, n. 07, p. 913-941, 2013.

[44] FIGUEIREDO, E. et al. On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. 2008 12th European Conference on Software Maintenance and Reengineering, 2008.

[45] ECLIPSE. AspectJ. https://eclipse.org/aspectj/. Acesso 10 set. 2015.

[46] BONDAVALLI, Andrea; BRASILEIRO, Francisco; RAJSBAUM, Sergio. Dependable Computing. Berlin: Springer, 2007.

[47] GRADECKI, Joseph D.; LESIECKI, Nicholas. Mastering AspectJ: Aspect Oriented-Programming in Java. United States: Wiley, 2003.

[48] KISELEV, Ivan. Aspect-oriented programming with AspectJ. Indianapolis, Ind.: Sams, 2003.

[49] ECLIPSE. Pointcuts. https://eclipse.org/aspectj/doc/next/progguide/semantics-pointcuts.html. Acesso 10 jan. 2016.

[50] ECLIPSE. AJDT. http://www.eclipse.org/ajdt/. Acesso 10 jan. 2016.

[51] GREENWOOD, P. et al. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. ECOOP 2007 – Object-Oriented Programming, p. 176-200, 2007.

[52] FERNANDES, Lyrene. AOV-graph Modeling to the HealthWatcher. http://www.cin.ufpe.br/~scbs/testbed/requirements/aov/HealthWatcher_AOV-graph.doc. Acesso 12 jan. 2016.

[53] KULESZA, U. et al. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. 2006 22nd IEEE International Conference on Software Maintenance, 2006.

[54] GARCIA, A. et al. Modularizing design patterns with aspects. Proceedings of the 4th international conference on Aspect-oriented software development - AOSD '05, 2005.

[55] APACHE TOMCAT. http://tomcat.apache.org. Acesso 14 jan. 2016.

[56] GOOGLE DEVELOPERS. Metrics. http://developers.google.com/java-dev-tools/codepro/doc/features/metrics/metrics. Acesso 14 jan. 2016.

[57] ORACLE. Implementing a JPA Named Query. http://rdocs.oracle.com/html/E13981_01/ent30qry001.htm. Acesso 15 jan. 2016.

[58] ORACLE. Annotation Type GeneratedValue. http://docs.oracle.com/javaee/7/api/javax/persistence/GeneratedValue.html. Acesso 15 jan. 2016.