INTRODUÇÃO

A programação orientada a objetos foi motivada devido a necessidade de desenvolvimento de software de qualidade, para uma maior reutilização e níveis de manutenção, e assim aumentar a produtividade e apoio para mudanças de requisitos no desenvolvimento de software (MEYER, 1997).

A programação orientada a objetos tem como um dos grandes diferenciais a definição de suas estruturas no conceito de herança, no qual facilita a extensão das classes já existentes na aplicação. Também se pode enfatizar a importância do polimorfismo, que permite ao programa utilizar funcionalidades dinamicamente durante a execução.

Baseado em uma programação modular em classes, a programação orientada a objetos disponibiliza facilmente a manutenção do sistema e reutilização de código, o que favorece o desenvolvimento de sistemas mais eficientes e complexos.

Para facilitar o processo de desenvolvimento de software são utilizados frameworks, os frameworks são códigos que possibilitam o compartilhamento de muitas características com técnicas de reutilização em geral na Programação Orientada a Objetos (JOHNSON, 1997).

Framework é uma das várias técnicas de reutilização no desenvolvimento de sistema, proporcionando componentes que podem ser acoplados facilmente a novos sistemas, também pode ser definido como uma base para o desenvolvimento de aplicações maiores e mais especificas, visto que o mesmo contém uma coleção de classes, técnicas, funções e metodologia para o desenvolvimento ágil e fácil de novos sistemas (JOHNSON, 1997).

Para o desenvolvimento de sistemas, não há necessidade do desenvolvedor conhecer como os componentes foram implementados, sendo necessário somente o entendimento da especificação destes componentes.

O Framework CodeIgniter é um conjunto de ferramentas para a construção de aplicações em PHP, e através do modelo MVC, permite desenvolver projetos com mais agilidade, utilizando várias bibliotecas para tarefas comuns, como a criação de uma interface simples e estruturas lógicas para acessar essas bibliotecas, possibilitando que seja mantido o foco no projeto, minimizando a quantidade de código necessário para o desenvolvimento de uma aplicação.

A estrutura criada para o Framework Codeigniter tem como objetivo fazer a comunicação entre as camadas da maneira simples, onde os dados são passados entre as camadas através de *arrays* ou objetos (GABARDO, 2012).

Atualmente os softwares estão ficando maiores e mais complexos, e assim fica cada vez mais difícil a manipulação dos modelos representados, dificultando mensurar as modificações e reutilizações das partes modeladas devido à relação contida entre os mesmos.

Na programação orientada a objetos existe uma dificuldade de implementação, no qual é necessária a intersecção e chamada de um código em diferentes situações no desenvolvimento, no qual a programação orientada a aspectos tem como objetivo resolver algumas ineficiências da programação orientada a objetos, aumentando a modularidade do código totalmente separado, com a característica de afetar diversas partes do sistema, conhecidos como interesses transversais (SOARES, 2004).

A complexidade de um aplicação é mensurada através da fragmentação das suas funcionalidades, determinando o comportamento do sistema, sendo a parte conhecida por requisitos gerais que fazem parte da implementação da aplicação como por exemplo custo, performance, confiabilidade, manutenção, portabilidade, custos operacionais entre outros. Estes comportamentos da aplicação são chamados de requisitos não funcionais ou interesses transversais (CYSNEIROS, 2001).

A programação orientada a objetos tornou o desenvolvimento de aplicações mais flexível e reutilizável, possibilitando maior agilidade e facilidade na manutenção, principalmente modularização das funcionalidades do sistema, onde é visualizada as propriedades do sistema como componentes.

Apesar de suas vantagens, a programação orientada a objeto tem apresentada algumas limitações, às vezes sendo necessário lidar com códigos que representam diferentes características, onde é necessário chamar códigos amarrados em uma simples solicitação, como o tratamento de interesses transversais em novas funcionalidade (SOARES, 2004).

Sendo assim, o objetivo deste projeto é, através do framework CodeIgniter, incluir o suporte à interesses transversais utilizando conceitos de programação orientada a aspectos, para interceptar a camada de controlador do modelo MVC.

Desta forma, serão exibidos os conceitos utilizados, no qual poderá ser utilizado este conceito em outros frameworks para aumentar a modularização e facilitar o processo de desenvolvimento de software.

CAPITULO 1 - PROGRAMAÇÃO ORIENTADA A OBJETOS

A metodologia para desenvolvimento de aplicações computacionais teve início da década de quarenta, entretanto, naquela época não existiam linguagens de programação, sendo as instruções gravadas diretamente ao nível de máquina. No início da década de cinquenta, com o surgimento das linguagens montadoras, houve um grande interesse na criação de ferramentas para auxiliar no desenvolvimento de aplicações computacionais (CARMO, 2001).

Ocorreram 03 etapas de desenvolvimento distintas até o final da década de 60. Estas etapas tinham como objetivo aprimorar os conceitos básicos de desenvolvimento das expressões aritméticas, comandos, matrizes, listas, pilhas e procedimentos. Assim surgiram as linguagens da primeira geração, chamadas FORTRAN I e ALGOL 58, e após alguns aprimoramentos, surgiram às linguagens de segunda geração, chamadas FORTRAN II, ALGOL 60, COBOL 61 e LISP, surgindo posteriormente às linguagens de terceira geração, chamadas PL/I, ALGOL 68 e PASCAL, onde nesta terceira geração surgiu também a linguagem Simula, conhecida como "mãe" das linguagens orientadas a objeto (CARMO, 2001).

A premissa básica da Programação Estruturada foi estabelecida em 1968, onde todo programa pode ser escrito através de uma união de comandos primitivos ou subprogramas, indiferente do seu nível de complexidade. Estavam disponíveis três estruturas de controle para a viabilização no desenvolvimento de aplicações [Mon94]:

- Comandos complexos executados sequencialmente;
- Comandos condicionais (if - then - else);
- Comandos de repetição ou laço (for, while e repeat ... until).

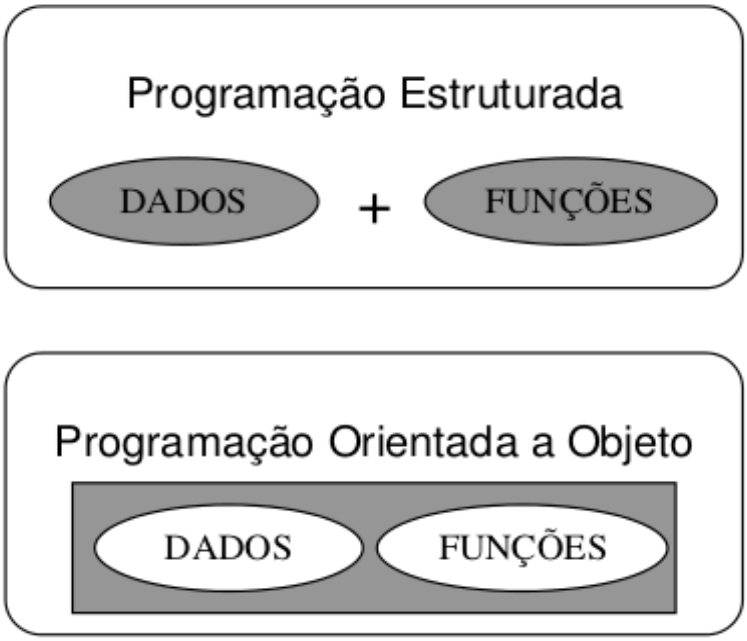
O desenvolvimento de diversos sistemas durante muitos anos seguiram este paradigma, mas a decomposição de aplicações em diversos subprogramas focava a atenção do desenvolvedor muito mais nos procedimentos do que nos dados, sendo este uma das principais fontes de críticas à Programação Estruturada.

Apesar do surgimento da programação orientada a objetos no final dos anos 70, a mesma passou a ser utilizada, em grande escala, como metodologia para desenvolvimento de softwares somente nos anos 90 (MARIANI, 1999).

O surgimento do paradigma de Programação Orientada a Objetos surgiu no começo da década de 60 com a Linguagem Simula, seguindo na década de 70 com a Linguagem Smalltalk e na década de 80 com a Linguagem C++ (CARMO, 2001).

Analisando os paradigmas de Programação Estruturada e a Programação Orientação a Objetos percebe-se que a grande diferença está na maneira de comunicação entre os dados e procedimentos da aplicação. No paradigma de Programação Estruturada o desenvolvimento é realizado através da comunicação de subprogramas e envio de dados, sendo assim, as aplicações são estruturadas através de procedimentos com o objetivo de manipular dos dados. No paradigma de Programação Orientada a Objetos os dados e procedimentos estão estruturados para a criação de classes de objetos, descrevendo um tipo de dado abstrato.

Figura 1- Elementos dos paradigmas da Programação Estruturada e Orientada a Objetos



Fonte: CARMO, Carla Soraia Leandro do. Automação de detalhamento de peças padronizadas em concreto armado via CAD e programação orientada a objetos. (Ano 2001, p.13)

Paradigma de Programação Orientação a Objetos se baseia na decomposição e classificação dos problemas, utilizando abstrações chamadas classes, sendo cada instância de uma determinada abstração a um objeto. Por usa vez, cada objeto pode ser considerado como um objeto do mundo real, sendo definido por uma interface que possua métodos que podem ser aplicados sobre o objeto (RODRIGUES, 2004).

A proposta da orientação a objetos é representar o mais fielmente possível as situações do mundo real nos sistemas computacionais. Entendemos o mundo como um todo composto por vários objetos que interagem uns com os outros. Da mesma maneira, a orientação a objetos consiste em considerar os sistemas computacionais não como uma

coleção estruturada de processos, mas sim como uma coleção de objetos que interagem entre si (FARINELLI, 2007).

Uma das principais diferenças da programação orientada a objetos, comparada com outros paradigmas de programação que disponibilizam a definição de estruturas e operações, é em relação ao conceito de herança, onde através deste conceito as estruturas existentes podem ser estendidas com facilidade. Além do conceito de herança pode-se enfatizar a importância do polimorfismo, que de forma dinâmica é possível alterar o comportamento que uma aplicação realizar, durante sua execução (RICARTE, 2001).

A programação orientada a objetos é fundamentada, pois engloba os princípios da abstração, hierarquização, encapsulamento, classificação, modularização, relacionamento, simultaneidade e persistência, no qual serão abordados nos próximos capítulos os principais conceitos deste paradigma.

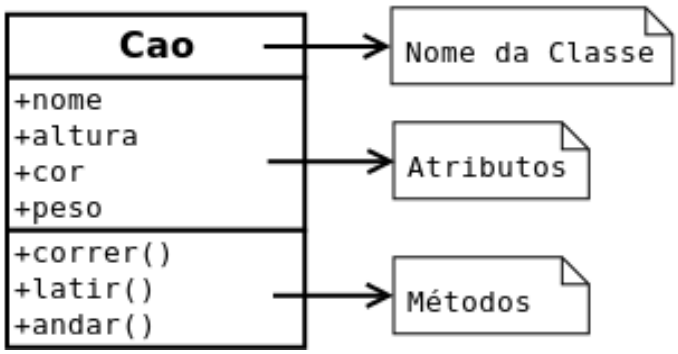
1.1 Classe

Considera-se uma classe a descrição de um conjunto de dados estruturados que tem como característica propriedades em comuns, também interpretada como uma estrutura modular completa que descreve de forma estática e dinâmica as propriedades dos elementos manipulados pelo programa (LASKOSKI, 2013).

Através da definição de uma classe são descritos todos os atributos dos objetos, e além da especificação destes atributos, esta definição também especifica as funcionalidades aplicadas aos objetos da classe, descrita através de métodos (RICARTE, 2001).

Na figura 2, pode-se observar um exemplo da definição de uma classe no paradigma orientado a objetos.

Figura 2- Definição da classe Cao



Fonte: Cláudio Rosse Pandolfi

Pode-se considerar assim que uma classe é definida por:

- Um nome da classe;
- Os nomes de seus atributos;
- Os nomes de suas funcionalidades descritas pelos métodos;

Ao realizar a análise entre dois cães, pode-se notar que ambos possuem o mesmo conjunto de atributos (nome, altura, cor, peso) e métodos (correr, latir, andar). Isso ocorre devido os dois cães fazem parte da mesma classe e, embora ambos possuam o mesmo conjunto de atributos e métodos, os valores definidos em cada atributo são diferentes entre eles (FARINELLI, 2007).

1.1.1 Atributos

Os atributos dos objetos são comparados com “variáveis” ou “campos” que armazenam valores diferentes, disponibilizando as características que os objetos podem conter (FARINELLI, 2007).

A descrição das propriedades de uma classe é realizada através de um conjunto de atributos, sendo cada atributo identificado por um nome e associado por um tipo.

O tipo associado a um atributo, no paradigma de programação orientada, é o próprio nome de uma classe, mas a maior parte das linguagens baseadas a objetos disponibilizam um grupo de tipos primitivos para serem utilizados na descrição dos atributos, como inteiro, real e caráter (RICARTE, 2001).

As mudanças dos valores dos atributos de um objeto são alterados somente através de estímulos externos ou internos e deverão ser manipulados exclusivamente por serviços ou método a própria classe. Assim, a única forma de modificar os valores associados aos atributos é através de disparos de eventos que estimulam a transição desses estados no objeto.

1.1.2 Métodos

As funcionalidades de uma classe são descritas através dos métodos, definindo qual será o comportamento dos objetos dessa classe.

Pode-se comparar os métodos como “procedimentos” ou “funções” que realizam as ações próprias do objeto. Todo o comportamento de uma classe é realizado através dos

métodos definidos, sendo possível que o objeto se manifeste e realize interação com outros objetos através de estímulos destes métodos (FARINELLI, 2007).

Cada método é especificado por uma assinatura, composta por um identificador para o método (o nome do método), o tipo para o valor de retorno e sua lista de argumentos, sendo cada argumento identificado por seu tipo e nome.

A comunicação entre objetos é realizada por estímulos, sendo requisitadas as ações através de mensagens entre eles, assim são executadas as rotinas responsáveis por acessar ou alterar os atributos dos objetos ou retornar algum comportamento do objeto.

1.2 Objeto

O objeto é caracterizado com uma instância de classes, possibilitando determinar quais informações um objeto contém e como serão manipuladas estas informações.

Pode-se interpretar um objeto como uma entidade capaz de armazenar as informações e oferecer várias operações para analisar ou para afetar estas informações.

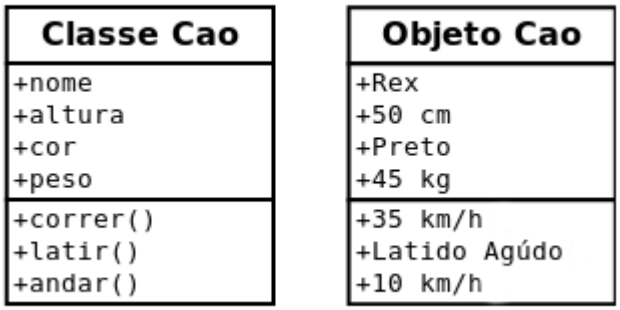
A classe é um elemento reconhecido em um texto de programa, mas um objeto é um conceito puramente dinâmico, o qual pertence à memória do computador e não ao texto do programa, ocupando espaço durante a execução (LASKOSKI, 2013).

O estado de um objeto é determinado quando um conjunto de valores é atribuído ao seus atributos durante a execução da aplicação, onde o comportamento do mesmo determina como será a ação e reação durante suas mudanças de estado e troca de mensagens com outros objetos.

Em linguagens de programação orientada a objetos, praticamente todo o processamento é realizado através dos objetos, atribuições de valores e comportamento na troca de mensagens entre eles (FARINELLI, 2007).

Ao utilizar a classe Cao, pode-se definir um valor aos atributos (nome, altura, cor e peso) e enviar mensagens ao objeto para analisar a ação e reação (correr, latir e andar).

Figura 3 - Diferença entre a classe Cao e o objeto Cao



Fonte: Cláudio Rosse Pandolfi

1.3 Abstração

A definição da palavra abstração no dicionário da língua portuguesa Michaelis, encontra-se o seguinte significado “Consideração das qualidades independentemente dos objetos a que pertencem. Processo pelo qual se isolam atributos de um objeto, considerando os que certos grupos de objetos tenham em comum”.

Considera-se abstrair o ato de separar elementos relevantes de um todo. Pode-se considerar que através da abstração são mapeadas as entidades em objetos da aplicação, sendo o produto final de um programa a representação de algum aspecto do mundo real (CARMO, 2001).

A abstração também pelo princípio que cada componente deve ocultar informações, disponibilizado somente o necessário e revelado o mínimo necessário para a realização das operações (FARINELLI, 2007).

Através do conceito de classes e objetos é possível realizar esta abstração, onde a decisão pelo conjunto correto de abstração é um fator determinante entre uma boa ou má análise orientada a objetos (CARMO, 2001).

Considera-se que o conceito de abstração, aplicado ao desenvolvimento de sistemas, a forma de representar somente o que será utilizado na aplicação, isolando os objetos que serão representados somente com as características relevantes. Para melhor entendimento, cita-se o seguinte exemplo:

Em um sistema de agenda telefônica, não tem relevância qual é a marca e modelo do aparelho utilizado, desta forma, esta informação não é inclusa na representação do objeto, mas em um sistema de gerenciamento de aplicativos para celular, esta informação passa a ser

relevante, visto que um aplicativo não funciona corretamente em determinado aparelho, sendo necessário incluir a representação da marca e modelo na construção do objeto.

O recurso da abstração pode ser utilizado como uma maneira de compreender problemas complexos, e diante deste problema, procurar dividi-lo em problemas menores, resolvendo individualmente cada um deles para solucionar o problema inteiro. Desta forma também é possível determinar maior e menor peso para as partes do problema, o que possibilita desconsiderar detalhes em determinados momentos para que outros sejam ressaltados (FARINELLI, 2007).

1.4 Encapsulamento

A implementação de dados correlacionados em um objeto ou entidade é conhecida, no paradigma de programação a objetos, como encapsulamento. O objetivo é possibilitar a reutilização de sistemas desenvolvidos sem dependência da implementação interna, e quando utiliza por outras entidades, existirão somente as interfaces de acesso aos membros das classes, desconsiderando como tais classes foram implementadas (CARMO, 2001).

“Encapsulamento é o princípio de projeto pelo qual cada componente de um programa deve agregar toda a informação relevante para sua manipulação como uma unidade (uma cápsula)”. (FARINELLI; Fernanda, 2007, p. 17).

O uso do encapsulamento e abstração torna a programação orientada a objetos um mecanismo poderoso, e recomenda que a representação de um objeto deve ser mantida oculta. Desta forma, cada objeto é manipulado através dos métodos públicos, pois somente a assinatura do objeto é revelada.

Na interação entre dois objetos é importante que um tenha conhecimento do conjunto de operações disponíveis do outro, criando uma interface operacional para que então envie e receba informação.

Ao criar a interface operacional, detalhes do processamento da operação do objeto não são conhecidos por outros objetos, o que possibilita ao usuário do objeto trabalhar em um nível mais alto de abstração, para que assim não haja a preocupação com os detalhes internos da classe encapsulada. Com isso, é possível simplificar a construção de aplicações mais complexas, tais como interfaces gráficas ou aplicações distribuídas (RICARTE, 2001).

Ao analisar a programação estruturada, o encapsulamento é uma grande vantagem da programação orientada a objetos, devido o mesmo disponibilizar todas as funcionalidades do

objeto sem que o usuário precise ter conhecimento das funcionalidades internas e de como é armazenada as informações deste objeto.

Para exemplificar pode-se analisar o envio de e-mails, onde simplesmente é redigido o e-mail e enviado, onde o usuário não quais os procedimentos realizados pelo provedor para que este e-mail seja enviado ao remetente.

No encapsulamento, outro ponto importante é a possibilidade de realizar modificações internas em um objeto para acrescentar novos métodos sem afetar os outros componentes utilizados por sistemas que utilizam este objeto (FARINELLI, 2007).

Ao analisar novamente o exemplo citado anteriormente, sabe-se que o provedor de e-mail pode realizar alterações no processo do envio de e-mail, sendo realizados ajustes de segurança, acrescentada melhorias ou novas funcionalidades, mas isso sem alterar a forma de envio deste e-mail.

1.5 Herança

O que torna a orientação a objetos única é o conceito de herança. Herança é um mecanismo que permite que características comuns a diversas classes sejam fatoradas em uma classe base, ou superclasse. A partir de uma classe base, outras classes podem ser especificadas. Cada classe derivada ou subclasse apresenta as características (estrutura e métodos) da classe base e acrescenta a elas o que for definido de particularidade para ela (RICARTE, 2001).

“Relação entre classes, de modo a permitir o compartilhamento de atributos e serviços semelhantes. Com a herança pode-se definir novas classes por extensão, especialização ou combinação de outras já definidas”. (CARMO; Carla Soraia Leandro do, 2001, p. 16).

O conceito de herança pode ser exemplificado através da própria genética, onde um filho herda as características genéticas dos pais e repassa suas características para seus filhos.

Para a programação orientada a objetos, o conceito de herança possibilita que uma classe obtenha ou transmita os atributos e métodos de outra classe para expandir ou especializar estas características de alguma forma (FARINELLI, 2007).

Pode-se dizer também que o conceito de herança é uma forma inteligente de reaproveitamento de código, devido a possibilidade de compartilhamento de atributos e métodos.

Desta forma, uma nova classe criada pode herdar os atributos e métodos de outra classe, implementando apenas as características específicas desta classe e reutilizando utilizando todas as características da classe principal.

A classe que obtém as informações também é conhecida como subclasse, já a classe que transmite suas características é conhecida como superclasse. Existem algumas formas de realizar os relacionamentos em herança, onde:

- **Extensão**: neste caso a superclasse é mantida inalterada e subclasse estende as características da superclasse, implementando novos atributos e/ou métodos.
- **Especificação**: a subclasse não implementa suas funcionalidades, sendo a superclasse que especifica o que será oferecido pela subclasse, através de um conjunto especifico de métodos públicos.
- **Extensão e Especificação**: a mais comum de ocorrer ao utilizar a programação orientada a objetos. Também conhecida como herança polimórfica, ao herdar as características da superclasse, a subclasse implementa alguns métodos da superclasse, redefinindo estes métodos para especializar o comportamento, sendo que os outros métodos terão o mesmo comportamento da superclasse.

Além da economia de código, a herança oferece maior integridade na manutenção da aplicação, uma vez que ao se alterar o comportamento em uma superclasse, as subclasses ligadas também serão alteradas e utilizarão os métodos atualizados sem necessidade de reprogramação (FARINELLI, 2007).

1.6 Polimorfismo

Polimorfismo é a capacidade de um objeto de se adaptar a diferentes tipos de dados, desta forma a aplicação é capaz diferenciar, dentre os métodos homônimos, qual será executado, devido a identificação do objeto receptor (CARMO, 2001).

Pode-se definir também como polimorfismo a possibilidade de uma variável ser referenciada a objetos de diversas classes, durante o tempo de execução da aplicação, ou defini-lo como a capacidade que objetos distintos têm de responder a uma mesma mensagem (FARINELLI, 2007).

Outra interpretação do polimorfismo é quando duas ou mais subclasses podem invocar métodos que têm a mesma identificação, conhecida também como assinatura do

método, mas o comportamento de cada subclasse é distinto, devido a especialização realizada (RICARTE, 2001).

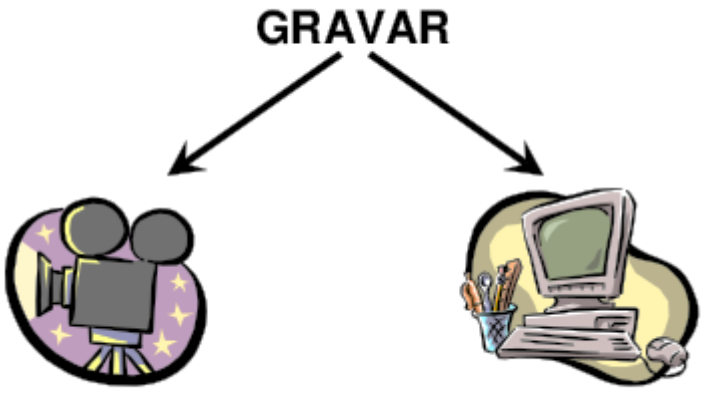
Fundamental na programação orientada a objetos, o polimorfismo possibilita definir comportamentos genéricos dos objetos, abstraindo detalhes particulares quando esses não forem necessários. Desta forma, uma mesma mensagem pode conter comportamentos diferentes durante a execução, próprias de cada objeto.

Para a utilização do polimorfismo, é necessário que uma classe defina os métodos exatamente com a mesma assinatura do método definido na superclasse, utilizando o mecanismo de redefinição de métodos, conhecido também como *overriding* (RICARTE, 2001).

Não tem como definir, através do compilador, qual método será utilizado, caso o método seja definido em outras classes. Desta forma a decisão de qual método será utilizado é definido durante o tempo de execução da aplicação, realizado pelo mecanismo de ligação tardia, conhecido também como *late binding*, *dynamic binding* ou *run-time binding*.

Para exemplificar o polimorfismo, pode-se identificar que o computador e a filmadora possuem uma funcionalidade com a mesma assinatura (gravar), mas o comportamento de cada funcionalidade é distinta entre os dos objetos (FARINELLI, 2007).

Figura 4 - Exemplo de Polimorfismo.



Fonte: FARINELLI, Fernanda. Conceitos Básicos de Programação Orientada a Objetos. (Ano 2007, p.23)

CAPITULO 2 - FRAMEWORK PARA DESENVOLVIMENTO

Como visto no capítulo anterior, uma aplicação orientada a objetos é desenvolvida por classes que, através de troca de mensagens, realizam determinado processo em um sistema.

Pelas características fornecidas pelo paradigma de orientação a objetos é possível reutilizar partes da aplicação, através de alguma interface que permita acesso aos serviços (caixa-preta) ou redefinindo o comportamento de algumas subclasses (caixa-branca). Desta forma, pode-se obter diferentes aplicações utilizando a base de uma aplicação já existente, reutilizando códigos ou projetos desta aplicação (CARMO, 1997).

Para esta reutilização pode ser necessário redefinir alguns comportamentos, onde a quantidade de modificações das classes depende do grau de semelhança entre as aplicações. No entanto, caso seja analisado a redefinição é possível identificar que existe uma parte comum de comportamento que podem ser generalizado, onde o programador pode criar superclasses que proporcionam um comportamento abstrato.

O conceito de classes abstratas é a representação genérica referente a determinadas entidades relacionadas, onde cada entidade representa uma abstração que será representada por uma subclasse concreta. Assim, a subclasse fornece as variações específicas do comportamento definido pela classe abstrata.

Como uma classe abstrata funciona como molde para subclasses, uma aplicação desenvolvida por classes abstratas pode ser utilizada como molde para a construção de outras aplicações. Neste caso, um projeto construído a partir de classes abstratas é considerado com um framework orientado a objetos (CARMO, 1997).

Os frameworks são considerados estruturas de classes inter-relacionadas, que permitem minimizar o esforço para o desenvolvimento de aplicações, devido a definição de sua arquitetura, o que desprende o desenvolvedor de software de implementar estas características.

“Um framework é um projeto de larga escala que descreve como um programa é decomposto em um conjunto de objetos interativos. É geralmente representado como um conjunto de classes abstratas e a maneira como suas instancias interagem”. (JOHNSON; Ralph E., 1997, p. 4).

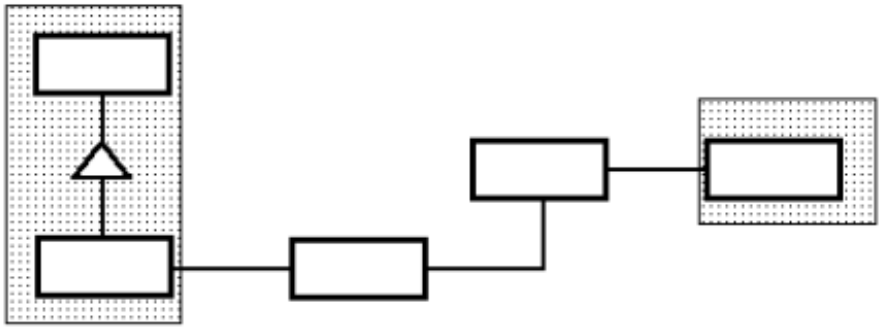
A utilização de frameworks invertem a modo de reutilização de classes, sendo iniciado o desenvolvimento através do entendimento do sistema contido no framework,

implementando somente as particularidades da aplicação específica. Desta forma, a aplicação é desenvolvida a pela adaptação da estrutura de classes do framework (TALIGENT, 1995).

Um framework se difere de uma reutilização de classes de uma biblioteca, pois em uma biblioteca são utilizados artefatos isolados de softwares, mas no caso do framework são reutilizados um conjunto de classes inter-relacionadas.

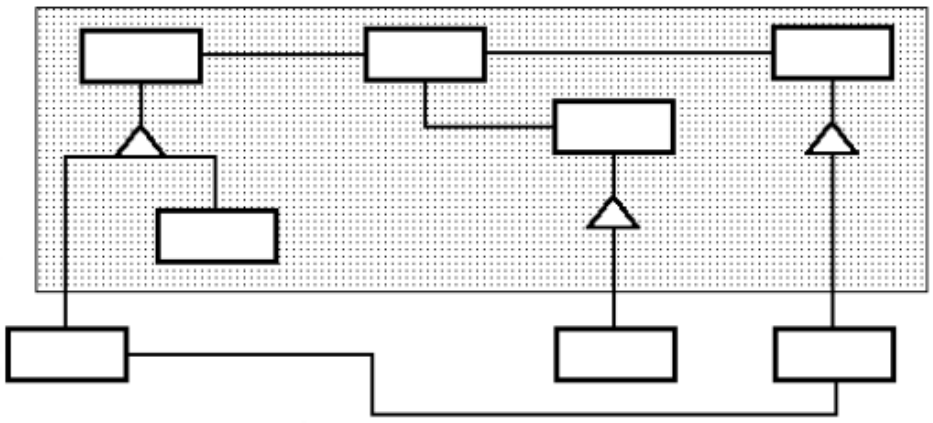
As figuras 5 e 6 ilustram a diferença entre a reutilização de classe de uma biblioteca e um framework, onde as partes sombreadas representam as classes associadas ao reuso (SILVA, 2000).

Figura 5 - Aplicação desenvolvida reutilizando classes de biblioteca.



Fonte: SILVA, Ricardo Pereira. Suporte ao Desenvolvimento e uso de Frameworks e Componentes, Instituto de Informática. (Ano 2000, p.31)

Figura 6 - Aplicação desenvolvida reutilizando um framework.



Fonte: SILVA, Ricardo Pereira. Suporte ao Desenvolvimento e uso de Frameworks e Componentes, Instituto de Informática. (Ano 2000, p.32)

Devido as facilidades que os frameworks oferecem, tais como o aproveitamento de código implementado, custo de desenvolvimento reduzido e uma estrutura comum e bem definida, muitas aplicações web são desenvolvidas a partir destes frameworks (SILVA, 2004).

Um framework é caracterizado por dois aspectos (TALIGENT, 1995):

- Fornecer uma infraestrutura de projeto, onde as interconexões preestabelecidas definem a arquitetura da aplicação, o que reduz a quantidade de código a ser desenvolvido, testado e depurado. Desta forma, o desenvolvedor fica livre das responsabilidades pré-definidas pelo framework, que é adaptado com as necessidades específicas da aplicação.
- Fornecer um fluxo de controle da aplicação onde, em tempo de execução, as instancias das classes desenvolvidas realizam ou aguardam as chamadas das instancias do framework.

Um framework também pode ser caracterizado pelo nível de granularidade, onde são agrupados diferentes tipos de classes mais ou menos complexas, podendo conter desde um projeto genérico completo até a construção de alto nível para solucionar situações comuns do projeto. Além disto, podem conter o projeto genérico completo para um domínio de aplicação, ou construções de alto nível que solucionam situações comuns em projetos (SILVA, 2000).

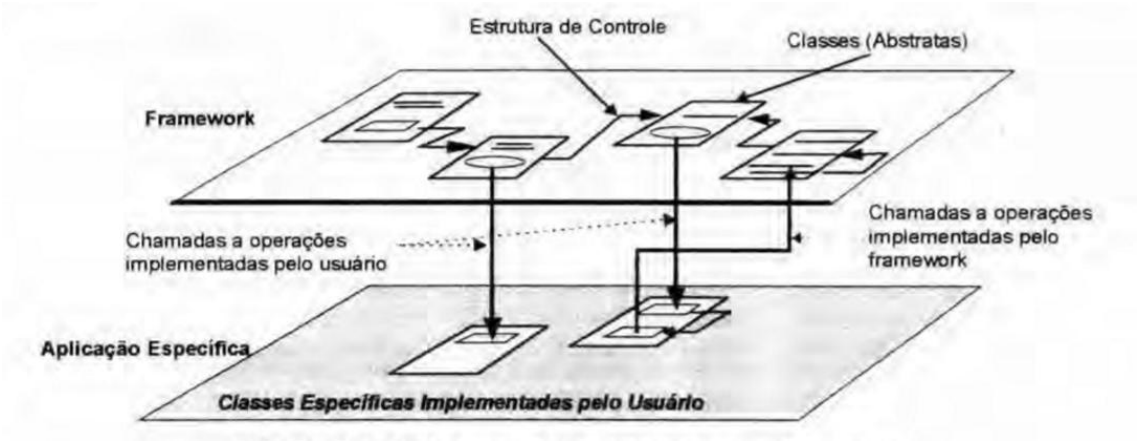
A vantagem de utilizar um framework é que os desenvolvedores utilizam as mesmas convenções, classes e bibliotecas, o que torna a manutenção de uma aplicação mais fácil. Assim, independente do desenvolvedor que criou determinado script, não é necessário horas para o entendimento e manutenção deste script por outro desenvolvedor. Desta forma, o framework contribui para que novos desenvolvedores possam, de forma rápida, compreender a aplicação implementada, e consequentemente diminui custos e tempo para treinamento (MINETTO, 2007).

Pode-se também considerar uma vantagem do framework a automatização de tarefas repetitivas. Considerando o fato de que em uma aplicação é necessário a manipulação dos dados de uma tabela no banco de dados, as operações de consulta, inclusão, exclusão e alteração são praticamente idênticas para todas as tabelas. Neste caso, não faz sentido para o desenvolvedor implementar este código diversas vezes, onde estas operações poderiam ser automatizadas por ferramentas contidas em um framework.

Através de um framework, a aplicação desenvolvida pode ser interpretada por dois níveis de composição, um nível superior e outro inferior. No nível superior, representado pelas classes do framework, fornece a estrutura de controle da aplicação, sendo o nível inferior contextualizado pelo desenvolvedor através de subclasses. Desta forma, o nível inferior

disponibiliza operações especificas de uma aplicação e a ativação destas operações é modelada através do nível superior (CARMO, 1997).

Figura 7 - Visão conceitual da estrutura de um framework.



Fonte: CAMPOS, Marcelo Ricardo. Compreensão Visual de Frameworks através de Introspeção de Exemplos. (Ano 1997, p. 32)

A partir da predominância de um projeto, o framework pode ser diferenciado em dois tipos:

- Frameworks Baseados em Herança: Denominado também como frameworks dirigidos pela arquitetura (architecture-driven frameworks) ou framework caixa branca, são disponibilizados para uma aplicação uma estrutura completa, sendo estendidas por códigos específicos, conforme a particularidade da aplicação. Neste caso, é necessário conhecer o funcionamento deste tipo de framework para a adaptação a aplicação que será desenvolvida (CARMO, 1997).
- Frameworks Baseados em Composição: Denominados também como frameworks dirigidos pelos dados (data-driven frameworks) ou frameworks caixa preta, são utilizados através da composição de objetos para sua adaptação. Desta forma, o framework tem comportamentos diferentes dependendo dos parâmetros que são enviado pelo cliente, sendo muito utilizados como base para a construção de toolkits de componentes (CARMO, 1997).

Estes dois tipos de frameworks são independentes, onde frameworks baseados em composição surgem de sucessivas generalizações de frameworks baseados em herança, devido ao desenvolvimento de um framework baseado em composição representar uma tarefa muito difícil e custosa (JOHNSON, 1993).

2.1 Arquitetura MVC

Com a necessidade de solucionar problemas apresentados pela programação monolítica, tais como o armazenamento de dados e códigos em uma mesma máquina, surgiu o conceito de desenvolvimento de sistemas através de camadas. Desta forma, solucionou-se a dificuldade de desenvolvimento e manutenção de aplicações, pois suas funcionalidades não ficavam em um módulo único que armazenava grandes quantidades de código.

Devido ao compartilhamento da lógica de acesso aos dados, impulsionou o desenvolvimento de aplicações em duas camadas, dividindo a base de dados da execução da aplicação, onde os armazenados dos dados eram realizados em uma máquina específica, e a aplicação executada em máquinas distintas.

Com a evolução da internet, surgiu uma nova visão de aplicação para os usuários e desenvolvedores, pois esta aplicação, vista como um site é disponibilizado através de um servidor web no qual apenas são realizadas requisições pelos usuários (MACORATTI, 2013).

Em consequência da evolução no desenvolvimento de software, novos padrões de projetos surgiram para facilitar a criação e manutenção de novas aplicações. O padrão de projeto MVC, um acrônimo para Model, View, Controller (em português Modelo, Visão e Controlador), teve grande destaque, sendo muito utilizado atualmente, onde tem o conceito de dividir a aplicação em três camadas distintas.

MACORATTI; José Carlos, (2013):

> “A arquitetura MVC - (Modelo Visualização Controle) fornece uma maneira de dividir a funcionalidade envolvida na manutenção e apresentação dos dados de uma aplicação. A arquitetura MVC não é nova e foi originalmente desenvolvida para mapear as tarefas tradicionais de entrada, processamento e saída para o modelo de interação com o usuário.”

O framework MVC surgiu através do ambiente Smalltalk-Visual Works com o objetivo de construir interfaces com o usuário. Sua representação na atualidade é uma das mais relevantes na estrutura de projeto adaptável dinamicamente, visto que sua fatoração de abstrações maximiza a reutilização de funcionalidades por composição de objetos. Pode-se considerar também que o grau de padronização de interfaces e encapsulamento de funcionalidade disponibilizam o desenvolvimento de interfaces de diálogos, o que possibilita

ao desenvolvedor descrever a composição da interface sem a necessidade de programação (CARMO, 1997).

Organizar o código em camadas é um dos principais objetivos do padrão MVC, o que acarreta a separação física dos componentes da aplicação, tornando essas camadas a chave para a independência entre os componentes, devido serem agrupados componentes com responsabilidades em comum (BALTHAZAR, 2006).

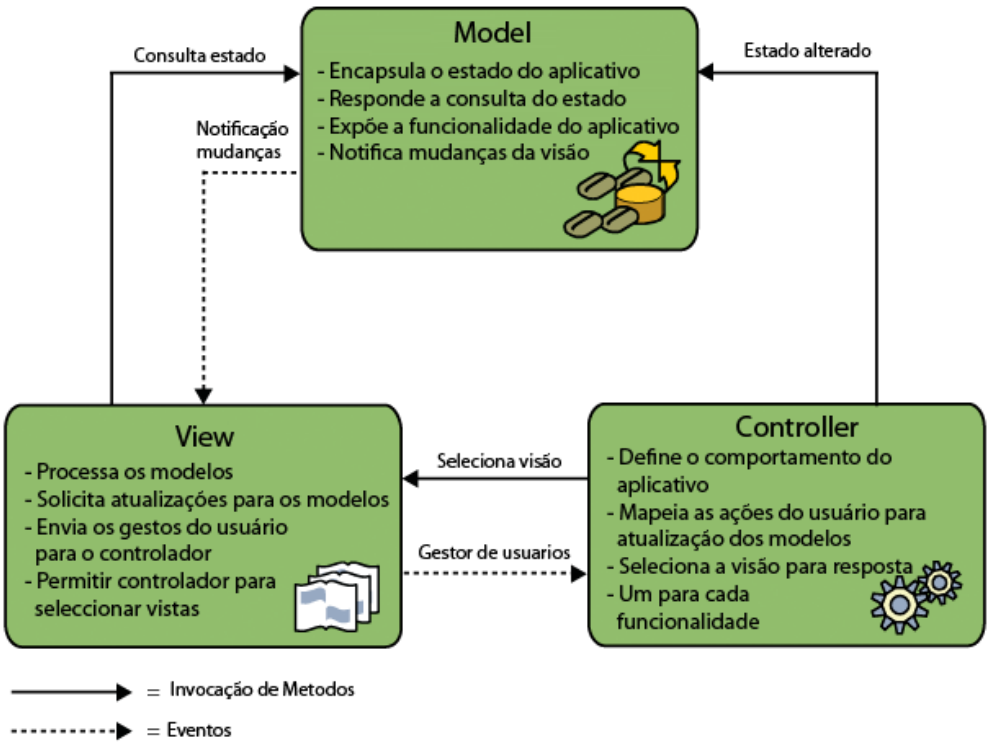
A aplicação desenvolvida pelo padrão MVC fica dividida em camadas, desta forma é proporcionado uma melhor manutenção, leitura e entendimento do código desenvolvido. Pode-se considerar também que aplicações que são desenvolvidas com este padrão se tornam mais independentes, devido a separação da camada de regra de negócio, da camada de armazenamento e da camada de visão, possibilitando realizar alterações na regra de negócio sem modificar a interface disponível para a usuário.

Com isso, as camadas de negócio podem ser divididas em classes, possibilitando agrupar estas classes em pacotes ou componentes, o que reduz as dependências entre as mesmas. Desta forma, esta divisão das classes proporciona a reutilização por partes distintas da aplicação, podendo ser utilizadas até por outra aplicação (BALTHAZAR, 2006).

As camadas são divididas, como se pode observar no nome do framework, em Modelo, Visão e Controlador, onde:

- Modelo (Model): camada que representa os dados e as regras específicas da aplicação (business data e business logic) para gerenciamento do acesso e a atualização desses dados.
- Visão (View): camada que gerencia a saída gráfica e textual da parte da aplicação visível ao usuário. Renderiza o conteúdo de um modelo acessando seus dados e especificando como eles são apresentados.
- Controlador (Controller): camada que interpreta as entradas de mouse e teclado do usuário, definindo o comportamento da aplicação. Envia as requisições do usuário para o modelo e seleciona as visões para apresentar os resultados destas requisições.

Figura 8 - Camadas do padrão MVC.



Fonte: https://netbeans.org/kb/docs/javaee/ecommerce/design.html

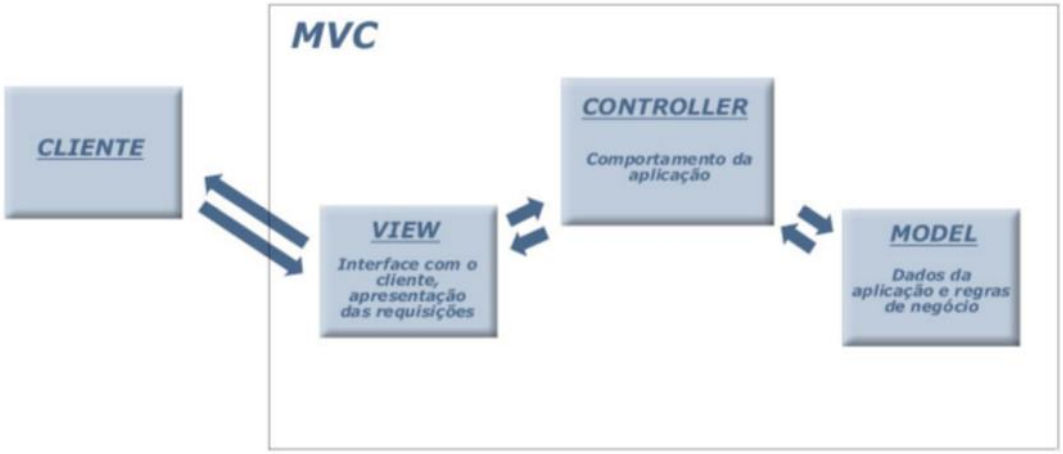
De forma conceitual, a instância de uma visão sempre é associado a uma representação de um objeto, através de um modelo, sendo associada opcionalmente por um controlador.

Através do controlador é realizada a coordenação das atividades providas na visão, as ações realizadas pelo usuário e o modelo correspondente que será instanciado. Além de representar a interface de saída da aplicação, a visão monitora as alterações produzidas através de anúncios de mudanças, gerados quando alguma modificação é realizada no modelo que interfere em seu estado interno, ocorrendo uma atualização da visão para manter consistente os dados entre os estados interno e de apresentação. Na associação, um modelo pode ter várias visões, mas uma visão pode ter só um modelo associado (CARMO, 1997).

Pode-se observar na figura 9, a separação física e lógica proposta pelo modelo MVC. Neste caso, é apresentada na camada de visão toda interface da aplicação que é interativa ao usuário, o que permite ao mesmo realizar requisições para a aplicação. Toda a parte de gerenciamento do comportamento da aplicação é desenvolvida na camada de controle, que realiza o intermédio na comunicação entre a visão e o modelo. A responsabilidade da camada de modelo é conter o código da requisição solicitada como, por exemplo, realizar a

comunicação com outra camada para realizar uma consulta no banco de dados (BALTHAZAR, 2006).

Figura 9 - O Modelo MVC.



Fonte: BALTHAZAR, Glauber da Rocha, GUIMARÃES, Fabio Mendes Ramos, PAULA, Melise Maria Veiga de, FILHO, Elio Lovisi. Uma Abordagem Prática sobre a Aplicação do Padrão MVC com o Framework Struts. (Ano 2006, p. 3)

Além da vantagem de fácil manutenção, devido ao sistema ser tratado como módulos específicos, outra vantagem no desenvolvimento de aplicações com o framework MVC é a divisão do acesso aos dados, regra de negócio e apresentação das informações. Desta forma, é possível fornecer um trabalho em equipe, pois enquanto o designer trabalha na apresentação das informações, um desenvolvedor pode, em paralelo, implementar as regras de negócio da aplicação, e um administrador de banco de dados pode criar os modelos da aplicação, visto que qualquer alteração entre as camadas teriam pouco ou nenhum impacto entre elas (GONÇALVES, 2007).

Ainda dentro do conceito de vantagens na utilização do framework MVC, pode-se considerar que a reutilização é fortemente explorada, visto que a mesma funcionalidade implementada em uma parte da aplicação pode ser reutilizada para outras partes da mesma aplicação (BALTHAZAR, 2006).

Porém, pode-se se considerar como desvantagem, na utilização do framework MVC, o custo elevado no desenvolvimento de uma aplicação, devido a curva de aprendizado na utilização do mesmo.

Isto ocorre devido todas as classes serem organizadas em pacotes, conforme a representação de seus comportamentos. Sendo assim, o desenvolvedor é obrigado a seguir um padrão, durante os ajustes ou desenvolvimento de novas funcionalidades, aumentando o prazo

para a finalização do produto final. Outra desvantagem que pode ser considerada é que este o framework MVC precisa ter um profissional especializado na equipe de desenvolvimento com domínio nos conceitos apresentados, o que pode aumentar os gastos na implementação da aplicação devido treinamentos e conscientização da efetiva adoção do padrão (BALTHAZAR, 2006).

2.2 Framework CodeIgniter

CodeIgniter é um framework para desenvolvimento de aplicações para quem desenvolve sites usando a linguagem PHP. O objetivo deste framework é permitir sejam desenvolvidos projetos mais rápido do que se fosse desenvolver uma aplicação com o código a partir do zero, através de um conjunto de bibliotecas para as tarefas mais comuns necessárias, bem como uma interface e estrutura lógica simples para acessar essas bibliotecas. CodeIgniter permite ao desenvolvedor mantenha o foco em seu projeto minimizando a quantidade de código necessário para uma determinada tarefa (CodeIgniter, 2013).

“CodeIgniter é um framework PHP de alto desempenho, com características de fácil aprendizado, construído para programadores PHP que necessitam de uma ferramenta simples e elegante para criar aplicações web com recursos completos”. (SITE <http://ellislab.com/codeigniter>; acesso em 19 de nov. de 2013).

CodeIgniter é baseado no padrão MVC. Conforme descrito na sessão anterior, o padrão MVC é uma abordagem de software que separa em camadas a regra de negócio, a manipulação dos dados e a da apresentação de uma aplicação, onde (CodeIgniter, 2013) :

- O modelo representa as suas estruturas de dados. Normalmente, sua classe de modelo conterão funções que ajudam a recuperar, inserir e atualizar as informações no banco de dados.
- A visão é a informação que está sendo apresentada a um usuário, onde será normalmente uma página web, mas em CodeIgniter, uma visão também pode ser um fragmento de página como um cabeçalho ou rodapé. Ela também pode ser uma página RSS, ou qualquer outro tipo de saída de dados para exibição ao usuário.
- O controlador atua como um intermediário entre o modelo, a vista e qualquer outro recurso necessário para processar a solicitação HTTP e gerar uma página web.

A estrutura do framework CodeIgniter é desenvolvida para realizar a comunicação entre as camadas do padrão MVC da forma mais simples possível, onde é realizada esta

comunicação enviando dados entre as camadas pelo uso de vetores, vetores multidimensionais ou objetos (GABARDO, 2012).

CodeIgniter tem uma abordagem bastante flexível do padrão MVC, onde os modelos não são necessários. Alguns desenvolvedores acreditam que não precisa utilizar a camada de modelo, ou achar que a manutenção de modelos na aplicação acarreta em maior complexidade do que se deseja, desta forma, pode-se dispensar a utilização de modelos e desenvolver aplicação minimamente usando controladores e visão (CodeIgniter, 2013).

CAPITULO 3 - PROGRAMAÇÃO ORIENTADA A ASPECTOS

O surgimento da programação orientada a objetos trouxe uma mudança na perspectiva de desenvolvimento de aplicações, desta forma, as aplicações se tornaram mais reutilizáveis, flexíveis e de fácil manutenção, possibilitando também o encapsulamento de funcionalidades especificas da aplicação em módulos.

Porém existem algumas limitações na programação orientada a objetos que são difíceis de serem implementadas, se considerar algumas limitações que são mais difíceis de serem tratadas utilizando o contexto de objetos. Estas limitações, conhecidas como interesses transversais, são representações de códigos entrelaçados e espalhados em diversas partes da aplicação, como por exemplo, a utilização do código de acesso ao banco de dados em diversas partes da aplicação (OSSHER, 1996).

Algumas abordagens podem ser consideradas para a separação destes interesses transversais, como por exemplo a programação orientada a sujeitos (OSSHER, 1996), orientação a aspectos (KICZALES, 1997) e separação multidimensional de interesses (OSSHER, 1999).

Estes conceitos representam uma melhoria no desenvolvimento de aplicações orientada a objetos, onde favorecem o desenvolvimento de aplicações, separando os interesses transversais em uma perspectiva além de classes e objetos. Desta forma, apesar da programação orientada a objetos conter abstrações e mecanismos muito útil, não são muito eficazes em modularizar os interesses transversais da aplicação (LOBATO, 2005).

Visando solucionar a modularização para a captura de algumas decisões de projeto, a programação orientada a aspectos tem demonstrado ser promissora, melhorando a qualidade de reutilização de código (HUGO, 2005).

A programação orientada a aspectos é uma das tecnologias utilizadas para a separação de interesses transversais e modularização de código. Através desta abordagem é possível fornecer ao desenvolvedor um conjunto de técnicas para a segregação de partes da aplicação em uma nova abstração, utilizando mecanismos de composição de aspectos e componentes (SILVA, 2008).

Desta forma, a programação orientada a aspectos aumenta a modularização da aplicação para separar os interesses transversais. Pode-se considerar em uma aplicação que interesses transversais são funções especificas que são utilizadas em diferentes partes do

sistema, como por exemplo, persistência de dados, distribuição, controle de concorrência, tratamento de exceções e depuração (HUGO, 2005).

Quando utilizado o paradigma de orientação a aspectos, é necessária a atenção dos analistas para especificar e separar os interesses transversais, onde os desenvolvedores devem entender que estes interesses transversais serão de responsabilidade tratadas nos aspectos.

Consequentemente, os desenvolvedores podem se dedicar em outras necessidades da aplicação devido a separação dos interesses transversais. Pode-se considerar também que criar equipes dedicadas ao desenvolvimento de aspectos, onde as mesmas trabalhariam em diversos projetos, enquanto outras equipes se dedicam ao desenvolvimento da regra de negócio da aplicação, utilizando os aspectos para compor os requisitos não funcionais da aplicação (HUGO, 2005).

Da mesma forma que outros paradigmas, a programação orientada a aspectos surgiu com uma linguagem de programação e atividades de implementação, surgindo extensões e ferramentas para este tipo de desenvolvimento.

Uma das principais extensões orientadas a aspectos é AspectJ, que provê aspectos para Java. Em AspectJ são disponibilizadas as abstrações fundamentais da programação orientada a aspectos, como os aspectos (aspects), pontos de junção (join points), *pointcuts* e *advices*.

Através dos aspectos é possível alterar a estrutura estática ou dinâmica de uma aplicação, tais como atributos e métodos de uma classe. Estas alterações são realizadas em tempo de execução da aplicação através dos pontos de junção, como por exemplo a chamada de um método. A descrição dos pontos de junção são realizadas pelos *pointcuts*, que disponibilizam várias formas de seleção, sendo associada a uma ação na interceptação dos pontos através de blocos chamados *advices*. Por sua vez, os *advices* definem quando o comportamento será executado, possibilitando que o mesmo seja executado antes, durante ou após os *pointcuts* (SILVA, 2008).

3.1 Interesse Transversal

Apesar da programação orientada a objetos ser uma tecnologia dominante no desenvolvimento de software, existem algumas funcionalidades que naturalmente afetam múltiplas classes ou métodos que modularizam outras funcionalidades da aplicação que não difíceis de serem implementadas utilizando o conceito de objetos (LOBATO, 2005).

Estas funcionalidades são conhecidas como interesses transversais ou *crosscutting concerns*. As características dos interesses transversais não possibilitam que sejam agrupadas em um módulo, o que acarreta o espalhamento deste código em diversas partes do sistema, o que torna a aplicação complexa e difícil de realizar o desenvolvimento e manutenção (LOUREIRO, 2005).

Na maioria dos conceitos de engenharia de software, os requisitos não funcionais não são representados adequadamente, pois são utilizados conceitos baseados em analise funcional ou orientada a objetos, que são limitadas para esta representação (ARAUJO, 2000).

Sendo um papel critico no desenvolvimento de aplicações, os requisitos não funcionais podem se tormar complexos em sua correção, devido a falta de representação ou a representação errada destes requisitos da aplicação (CYSNEIROS, 2001).

Estes conceitos de engenharia de software não disponibilizam suporte para esse tipo de requisito devido a dificuldade na representação dos mesmos, onde pode-se considerar algumas questões importantes, como uma restrição relacionada a uma solução de projeto, a tendencia de um requisito se relacionar a mais de um requisito funcional e a identificação destes requisitos não funcionais na aplicação (ARAUJO, 2000).

Sendo assim, os interesses transversais aumentam a dificuldade na implementação de uma aplicação de forma otimizada, pois são inseridas nos requisitos básicos do software, onde impossibilita separar estas funcionalidades em um componente ou em uma classe.

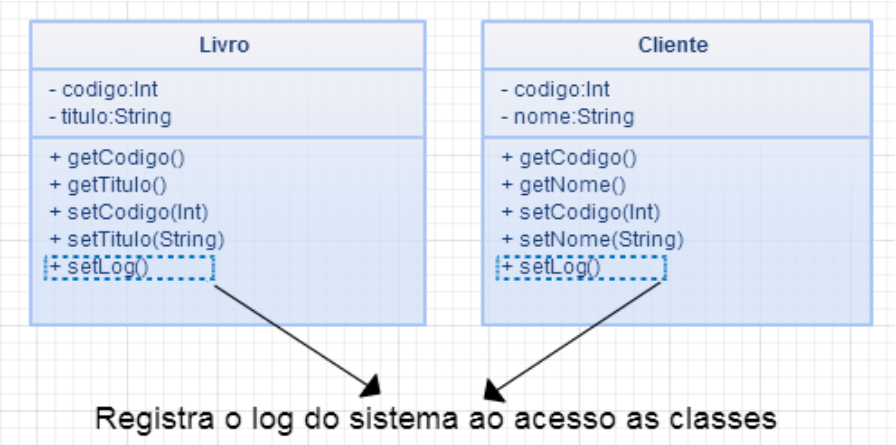
Existem vários tipos de requisitos básicos da aplicação que podem ser considerados como interesses transversais, tais como:

- Sistema de logs de aplicação, tornando o log simples e independente do sistema.
- Captura de informações durante a execução da aplicação, utilizando estas informações posteriormente para a análise de performance.
- Gestão de sessões, analisando de rastreamento e expiração desta sessão.
- Controle de tratamento de erros.
- Desenvolvimento de persistência de dados da aplicação.

Devido vários métodos ou funções terem dependências no uso dos interesses transversais, nem sempre estas funcionalidades se comportam da mesma maneira, o que impossibilita a junção destas funcionalidades em um módulo independente, evitando a chamada do código em diversas partes da aplicação (LOUREIRO, 2005).

Para entender o conceito de interesses transversais, pode-se observar na Figura 10 a representação UML de duas classes de uma aplicação.

Figura 10 - Exibição do registro de log para duas classes da aplicação.



Fonte: Cláudio Rosse Pandolfi

A Figura 10 representa uma estrutura simples para geração de log, um interesse transversal que normalmente é espalhado e entrelaçado em diversas partes da aplicação. As classes Livro e Cliente representam partes da aplicação que compartilham desta geração de log, através do método *setLog()*.

A funcionalidade do método setLog() pode ser distinta entre as classes Livro e Cliente, o que pode tornar a representação deste método distinto para cada classe, devido algumas particularidades ao armazenar o log dos objetos.

Neste caso, poderia se criar uma nova classe, separando o comportamento do log da aplicação através de uma nova classe denominada Log. Entretanto, esta separação não estará resolvendo a situação, pois permaneceria a chamada para o registro do log da aplicação nas classes Livro e Cliente.

3.2 Aspectos (Aspects)

Aspecto é considerada, em programação orientada a aspectos, a abstração que disponibiliza suporte para o isolamento de interesses transversais, ou seja, um aspecto é a correspondência de um interesse transversal e compõe a unidade modular projetada para interceptar determinadas classes e objetos da aplicação.

Um aspecto contém três propriedades básicas (Kiczales et al., 1997; Chavez et al., 2001; Elrad et al., 2001):

- Segregação baseada em aspectos: representa a partir da aceitação de uma diferença entre classes e aspectos, tendo uma distinção clara entre eles. Desta forma, como a

aplicação é desenvolvida através de aspectos, os aspectos modularizam os interesses transversais, enquanto as classes modularizam os outros interesses da aplicação.

- Inconsciência: propriedade onde o componente não precisa ser ajustado para ser interceptado por um aspecto, desta forma as classes afetadas pelo aspecto não percebem a interceptação não desejável da programação orientada a aspectos.
- Quantificação: possibilidade de implementar declarações únicas e separadas que afetam diversos pontos da aplicação, onde seja possível fazer uma declaração de determinado tipo em uma aplicação, consequentemente sempre que houver determinada condição, será executada uma ação.

Várias classes ou objetos podem ser afetados de diversas maneiras por um aspecto, interferindo na estrutura estática ou dinâmica destas classes ou objetos.

Um aspecto afeta a estrutura estática de uma classe/objeto através de declarações de intertipo, onde a estrutura dinâmica é interferida através de conjuntos de junção e *advices*.

3.2.1 Pontos de Junção (Join Points)

Os pontos de junção são considerados como pontos na execução de uma aplicação e representa um conceito fundamental da programação orientada a aspectos, sendo um elemento crucial na semântica da linguagem.

Na programação orientada a aspectos, os pontos de junção tem o papel de estabelecer a ligação entre uma determinada parte da aplicação e a execução dos *advices*, sendo este processo conhecido como *weaving*.

As identificações de casos que utilizam os pontos de junção durante o tempo de execução da aplicação são:

- Execuções de métodos ou construtores.
- Invocação de métodos ou construtores.
- Execução de *advices*.
- Acesso de leitura ou de escrita em um atributo.

3.2.2 Conjunto de Junções (Pointcut)

Considera-se um conjunto de junções um subconjunto de todos os pontos de junção dos aspectos disponíveis da aplicação, sendo este conjunto de junções especificado através de

um predicado (query). Assim, esta especificação constitui o primeiro passo na construção da abstração do aspecto, definindo quais pontos de junção irão atuar no aspecto durante a execução da aplicação.

Um conjunto de junção também especifica onde os pontos de junção serão executados. Este conjunto de junções é descrito por um nome e um corpo. Desta forma é possível alinhar diversos pontos de junção através de operadores lógicos (&& ou and, || ou or, ! ou not) (SILVA, 2006).

3.2.3 Adendo (Advices)

Os adendos são considerados construtores especiais que se assemelham com um método de uma classe, definindo o comportamento dinâmico que será executado quando é alcançado algum conjunto de junções definidos na aplicação. Pode-se dizer então que adendos são recursos que afetam dinamicamente o comportamento de classes e objetos (LOBATO, 2005).

Um adendo depende dos pontos de junção e do conjunto de junções para serem chamados durante a execução da aplicação, onde é invocado apenas quando um conjunto de junções é atingido (LOUREIRO, 2005).

Pode-se utilizar de várias maneiras um adendo na aplicação, sendo executado antes (before), ao redor (around), depois (after), depois do retorno (after returning) ou depois de executado (after throwing).

3.2.4 Declaração de Inter-tipos (Inter-type Declaration)

As declarações de inter-tipo são considerados simplesmente como uma adição às classes existentes na aplicação. Desta forma, é possível adicionar novas declarações às classes e objetos, possibilitando ajustar as necessidades da aplicação em relação a um aspecto (LOUREIRO, 2005).

Diferente de um adendo, que são executados em tempo de execução, as declarações de inter-tipos operam estaticamente, durante a compilação da aplicação, onde são adicionados novos atributos e/ou métodos através de declarações de inter-tipos nas classes que são interceptadas por um aspecto, ou modificado o relacionamento entre classes e/ou interfaces.

3.2.5 Tecelagem (Weaving)

Esta operação pode ser considerada como a mais importante e a mais complexa de completar. A tecelagem é importante, pois é nesta fase que os adendos, que estão presentes nos aspectos, são adicionados nos pontos de junção correspondentes.

A representação da tecelagem, na programação orientada a aspectos, é considerada um grande desafio. Esta ação é realizada através de ferramentas designadas geralmente por conjuntos de junção, para que não seja especificado manualmente pelo desenvolvedor.

Conforme citado por Kiczales (1997), juntamente com a sua equipe, conceberam na introdução da programação orientada a aspectos as seguintes hipóteses para o processo de tecelagem:

- Um pré-processamento, semelhante ao utilizado na implementação de C++.
- Um pós-processamento, combinando os arquivos binários resultantes.
- Um compilador com suporte a implementações de orientação a aspectos, gerando arquivos binários com as informações de tecelagem.
- Efetuar a tecelagem em tempo de carregamento.
- Disponibilizar a aplicação e executar a tecelagem em tempo de execução.

3.3 Ferramentas e Linguagens Orientadas a Aspectos

Atualmente existem algumas ferramentas e linguagens que possibilitam trabalhar com o paradigma de orientação a aspectos. A seguir serão listadas algumas destas ferramentas e linguagens.

3.3.1 QIDL

QIDL é uma extensão de CORBA, e tem o objetivo de funcionar através de definições de qualidade de serviço, onde são projetadas para realizar suporte a várias restrições diferentes. O compilador QIDL auxilia o usuário na implementação de serviços de QoS.

Desta forma, ao invés da utilização de um combinador de aspectos, os elementos de QoS são implementados através de um framework do compilador IDL. Os aspectos são gerados em IDL e o desenvolvimento é realizado na própria linguagem de componentes.

3.3.2 AOP /ST

A AOP/ST é uma ferramenta que disponibiliza nas linguagens Visualworks / Smalltalk suporte à programação orientada a aspectos. Esta ferramenta é composta de uma combinação do Smalltalk e duas linguagens de aspectos, onde uma linguagem é para o sincronismo de processos e outra para acompanhar o fluxo durante o tempo de execução da aplicação (SILVA, 2006).

3.3.3 AspectJ

AspectJ é uma extensão que disponibiliza suporte à programação orientada a aspectos para a linguagem Java de maneira genérica. O código desta extensão é aberto (Open Source), disponível a partir da versão 0.7b4, e possui suporte a ambientes integrados de desenvolvimento, sendo o projeto desta linguagem voltada através de respostas dos usuários em relação ao ambiente.

A implementação do combinador de aspectos é realizado através de um pré-processador, onde o mesmo recebe como entrada as aplicações de componentes e os aspectos, gerando os arquivos Java da aplicação final.

3.3.4 D

D é um framework que disponibiliza abstrações para o desenvolvimento de esquemas para o sincronismo e transferência de dados remotos em sistemas distribuídos. Para isso, este framework fornece duas linguagens para a descrição de aspectos:

- COOL: disponibiliza especificar mecanismos de exclusão mútua de maneira separada da aplicação.
- RIDL: disponibiliza mecanismos de manipulação de dados entre diferentes espaços de execução, funcionando separadamente dos componentes funcionais.

Através dessas linguagens, é possível integrar outras linguagens orientadas a objeto, sendo poucas as alterações realizadas na linguagem escolhida. Através do framework D, os aspectos descritos afetam o comportamento distribuído e concorrente dos componentes, desta forma o desenvolvedor primeiramente desenvolve as funcionalidades dos componentes e a

seguir, de forma explícita, o desenvolvedor descreve como os aspectos irão afetar os componentes, separado do restante da aplicação.

CAPITULO 4 - SUPORTE DE INTERESSES TRANSVERSAIS PARA FRAMEWORK CODEIGNITER

Os frameworks disponibilizam atualmente várias técnicas e padrões que facilitam no desenvolvimento de aplicações, sendo definido também como uma base no desenvolvimento de um novo software.

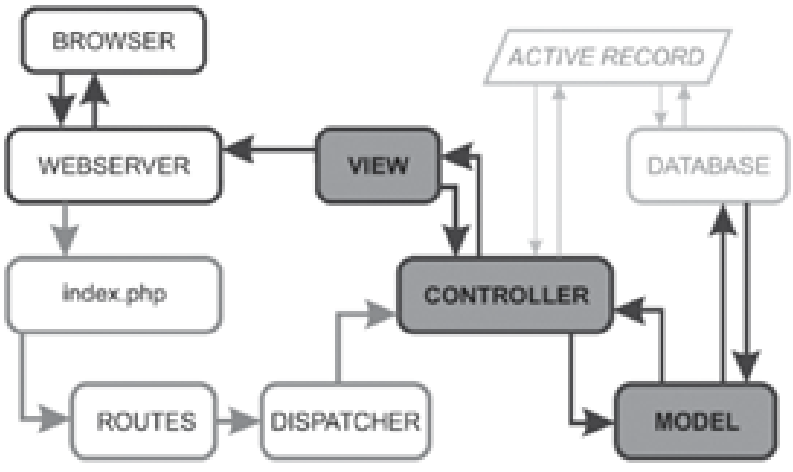
O framework CodeIgniter é um conjunto de ferramentas para a construção de aplicações em PHP, e através do modelo MVC, permite desenvolver projetos com mais agilidade.

Utilizando várias bibliotecas para tarefas comuns, como a criação de uma interface simples e estruturas lógicas para acessar essas bibliotecas, o desenvolvedor mantém o foco no projeto, minimizando a quantidade de código necessário para o desenvolvimento de uma aplicação.

A estrutura criada para o framework CodeIgniter tem como objetivo fazer a comunicação entre as camadas da maneira mais simples, onde os dados são passados entre as camadas através de *arrays* ou objetos (MEYER, 1997).

Através da Figura 11, pode-se observar como é a estrutura MVC e o fluxo de dados entre suas camadas.

Figura 11 - Estrutura MVC e fluxo de dados do framework CodeIgniter.



Fonte: GABARDO, Ademir Cristiano. PHP e MVC com Codeigniter. (Ano 2012, p. 21)

Para entender a estrutura ilustrada na Figura 11, nota-se que a camada do controlador é responsável por todas as ações provenientes do usuário, onde recebe, processa e retorna os dados para a visão.

Quando é realizada a requisição HTTP para o servidor web, sempre é propagada a esta requisição para o arquivo *index.php*, onde o mesmo é responsável pelo processamento da requisição (GABARDO, 2012).

Durante o processamento da requisição, o framework verifica se existe alguma rota correspondente e as informações em *cache* da requisição encaminhada, sendo disparadas para o controlador correspondente.

O controlador, por sua vez, processa a informação, requisitando ou enviando informações a um modelo, e após o processamento, carrega uma visão que poderá receber ou não informações providas pelo controlador, onde a visão é exibida como saída para o usuário (GABARDO, 2012).

No padrão MVC do framework CodeIgniter é possível que a camada de controle acesse diretamente o banco de dados sem a utilização da camada de modelo, utilizando uma classe chamada Active Record, contudo não é uma abordagem recomendada (MEYER, 1997).

Apesar de disponibilizar o padrão MVC, o framework CodeIgniter não disponibiliza o suporte a funcionalidades que afetam diversos objetos da aplicação, conhecidos como interesses transversais.

Sendo assim, aumenta a dificuldade na implementação destas funcionalidades de forma otimizada, pois é necessário inserir estas funcionalidades nos requisitos básicos da aplicação, impossibilitando separar estas funcionalidades em um componente ou em uma classe.

Existem vários tipos de requisitos básicos da aplicação que podem ser considerados como interesses transversais, tais como:

- Sistema de logs de aplicação, tornando o log seja simples e independente do sistema.
- Captura de informações durante a execução da aplicação, utilizando estas informações posteriormente para a análise de performance.
- Gestão de sessões, analisando de rastreamento e expiração desta sessão.
- Controle de tratamento de erros.

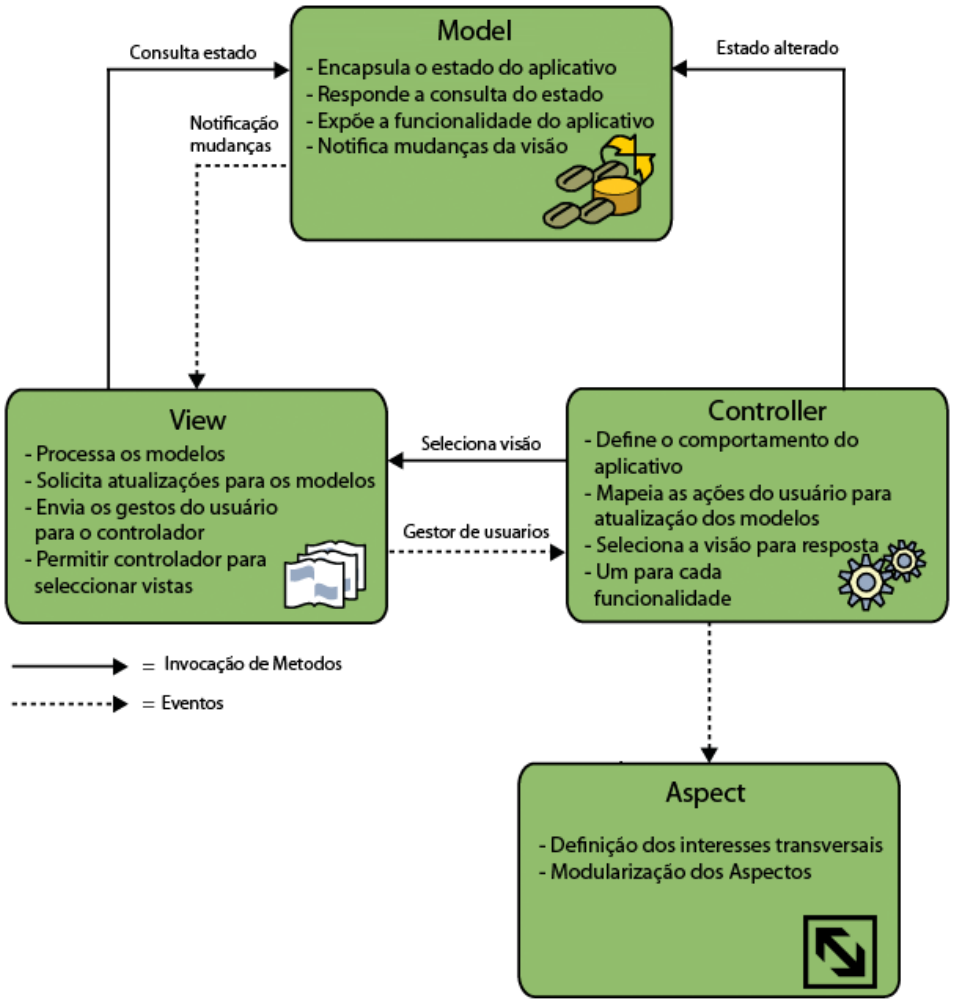
Através do paradigma de programação orientada a aspectos consegue-se resolver a limitação, contida na programação orientada a objeto, para suporte a interesses transversais.

Para o desenvolvimento do projeto será criado uma camada no framework

CodeIgniter para interceptação dos métodos executados no controlador do framework, utilizando o conceitos do paradigma de programação orientada a aspectos.

A Figura 12 exibe a camada de aspectos que será implementada para auxiliar na modularização, junto com o modelo MVC do framework CodeIgniter.

Figura 12 - Representação do Suporte a Interesses Transversais com o modelo MVC.



Fonte: http://www.macoratti.net/vbn_mvc.htm

Como resultado, será criada uma camada para suporte a interesses transversais, para a definição e modularização dos aspectos da aplicação, sendo realizada a comunicação com a camada de controle do framework, possibilitando uma melhor eficiência no desenvolvimento de aplicações.

Para os testes será desenvolvida uma aplicação para acesso ao sistema e o gerenciamento de cadastro de alunos, e através da camada para suporte a interesses transversais, serão criados os aspectos de log e gerenciamento de sessão da aplicação.

Para a finalização do projeto está previsto a realização de alguns ajustes diretamente nos aspectos, para verificar a modularização e manutenção da aplicação, e assim alcançar os resultados esperados.

4.1 Construção do Suporte de Interesses Transversais

No framework CodeIgniter, toda a requisição enviada para a aplicação é propagada para o arquivo *index.php*, sendo este arquivo responsável por verificar se existe alguma rota correspondente e as informações em *cache* da requisição, para assim enviar a requisição para o controlador correspondente.

No arquivo *index.php* são definidas algumas constantes de configuração do framework, possibilitando informar por exemplo se é utilizado um ambiente de desenvolvimento, teste ou produção e as pastas do sistema e da aplicação.

Para a inicialização do CodeIgniter, é carregado o arquivo *CodeIgniter.php*, localizado no núcleo (pasta core) do framework, onde é carregada as funções e métodos para desenvolvimento da aplicação, possibilitando trabalhar com o padrão MVC.

Através do arquivo *CodeIgniter.php*, são carregadas todas as classes da aplicação e realizada a chamada do controlador correspondente, enviando as informações da requisição.

Para incluir o suporte a interesses transversais no framework, são carregadas as classes contidas na pasta *aspects*, encontrada na pasta da aplicação (*application*), desta forma é possível localizar facilmente as classes de aspectos na aplicação.

Para a identificação de uma classe de aspectos é utilizado o conceito de anotações, conhecida como *annotations*. Anotações são utilizadas como marcações predefinidas, que através de um comentário no código da classe é possível incluir recursos adicionais na aplicação.

Para a criação dessas anotações, é utilizado o projeto Addendum, que através da API de reflexão da linguagem PHP, disponibiliza suporte a anotações com valores simples e múltiplos.

A versão 5 do PHP disponibiliza uma API completa de reflexão que adiciona a capacidade de classes, interfaces, funções, métodos e extensões de engenharia reversa. Além

disso, a API de reflexão oferece maneiras de recuperar comentários de documentação para as funções, classes e métodos.

Desta forma foi criado as anotações no arquivo *aspectAnnotation.php*, para predefinir as anotações do aspecto.

Para a identificação da classe como um aspecto, foi criada a anotação *Aspect*, sendo assim, quando criada uma classe de aspecto, é necessário inserir a anotação acima do nome da classe, conforme pode-se observar na Figura 13:

Figura 13 - Representação do Aspecto No Framework CodeIgniter.

```
/**
* @Aspect
*/
class Log_aspect {

...

}
```

Fonte: Cláudio Rosse Pandolfi

A identificação do conjunto de junções é representada através da anotação *Pointcut*. Nesta anotação é representado por um valor simples *execution*, informando os pontos de junção do aspecto. Esta representação é realizada através de uma expressão que informa o nome do controlador, método e parâmetros que representa o ponto de junção, onde podem ser utilizados alguns caracteres para representar os pontos de junção.

Para construir a expressão cada ponto de junção é representado por um conjunto com o nome do controlador, o método que será interceptado e os parâmetros contidos no método, e caso seja realizada uma chamada com estas características, o aspecto é executado durante a fase de execução da aplicação.

Figura 14 - Representação do Ponto de Junção

```
Pointcut(execution='NomeControlador método(parametro1,parametro2)')
```

Fonte: Cláudio Rosse Pandolfi

Para generalizar a interceptação de uma classe, é possível utilizar na expressão o caractere asterisco ("*"), desta forma pode-se realizar a interceptação de métodos com o mesmo nome em diversos controladores ou interceptar um controlador independentemente do método executado.

Figura 15 - Ponto de Junção para Interceptação do Método

```
Pointcut(execution='* método(*)')
Pointcut(execution=''controlador *(*)')
```

Fonte: Cláudio Rosse Pandolfi

Além da generalização na criação da expressão do conjunto de junções, pode-se representar diversos pontos de junção em uma mesma expressão, onde é utilizado duas barras verticais ("||"), idêntico ao operador logico "ou". Assim é possível representar vários pontos de junção para a execução do aspecto, conforme pode-se observar na Figura 16.

Figura 16 - Representação de Dois Pontos de Junção

```
Pointcut(execution='* método(*) || controlador *(*)')
```

Fonte: Cláudio Rosse Pandolfi

Devido a generalização da expressão, existe também a representação de um ponto de junção que pode não fazer parte dessa regra, desta forma, o aspecto é executado conforme definição do conjunto de junções, definindo uma exceção nesta execução. Para isso é utilizado o símbolo de exclamação ("!"), impedindo que determinado ponto de junção seja executado na interceptação do aspecto, pode-se observar a figura 17.

Figura 17 - Representação de Exceção de um Ponto de Junção.

```
Pointcut(execution='controlador *(*) || ! NomeControlador método(*)')
```

Fonte: Cláudio Rosse Pandolfi

A anotação *Pointcut* sempre é representada acima do método do aspecto que será executado, sendo assim, toda vez que for executado um ponto de junção representado no conjunto de junções, o método do aspecto será executado.

Na Figura 18 pode se observar a representação do conjunto de junções no framework CodeIgniter, para a execução do método *registraLog()* toda vez que é chamado o método *doLogin()* do controlador *Login*.

Figura 18 - Representação do Conjunto de Junções no Framework CodeIgniter.

```
/**
* @Pointcut(execution='Login doLogin(*)')
*/
function registraLog() {
    ...
}
```

Fonte: Cláudio Rosse Pandolfi

Para a interceptação do método executado pelo controlador, além de informar os pontos de junção, é necessário informar o adendo (*advice*). Este adendo pode ser representado pelas anotações:

- *Before*: executa o aspecto antes da chamada do controlador;
- *After*: executa o aspecto depois da chamada do controlador;
- *AfterReturning*: executa o aspecto após o retorno do controlador;
- *AfterThrowing*: executa o aspecto após um erro no controlador.

Assim como a anotação *Pointcut*, as anotações *Before*, *After*, *AfterReturning* e *AfterThrowing* sempre são representadas acima do método do aspecto que será executado, podendo informar diversos adendos no método do aspecto. Na Figura 19, pode-se observar a chamada dos adendos *Before* e *AfterReturning* para a execução do método *registraLog()*.

Figura 19 - Representação do Adendo no Framework CodeIgniter.

```
/**
* @Pointcut(execution='Login doLogin(*)')
* @Before @AfterReturning
*/
function registraLog() {
    ...
}
```

Fonte: Cláudio Rosse Pandolfi

Através das anotações criadas é possível identificar as características básicas da representação do aspecto, e assim possibilitar a execução de um método do aspecto em determinado momento da execução da aplicação. Assim, os pontos de junção são separados e executados conforme as anotações dos adendos inseridos no aspecto.

A execução deste método é realizada através da função *call_aspect()*, criada no arquivo *Common.php* localizado no núcleo do framework CodeIgniter. Esta função verifica a representação do conjunto de junções, e caso seja executado algum ponto de junção da aplicação, é realizada a chamada do método do aspecto através da API de reflexão.

Para realizar a chamada, após identificar um ponto de junção, é instanciada uma classe através da API de reflexão e recuperado o método para a invocação, enviando um *array* com os parâmetros da chamada do ponto de junção, conforme pode-se observar na Figura 20.

Figura 20 - Invocação do aspecto através da API de reflexão.

```
$rfClass = new ReflectionClass("nome_da_clase");
$mtdClass = $rfClass->getMethod("nome_do_metodo");
$mtdClass->invoke($rfClass->newInstance(),$parametros);
```

Fonte: Cláudio Rosse Pandolfi

A função *call_aspect()* é chamada após carregar as classes do controlador, executando os pontos de junção representados pela anotação *Before*, desta forma o aspecto é executado antes da execução do método do controlador.

Os pontos de junção representados pela anotação *AfterReturning* são executados realizando a chamada da função *call_aspect()* após o retorno da execução do controlador.

Para a execução do aspecto depois da execução do controlador, foi adicionada a chamada da função *call_aspect()* no final da execução da requisição do controlador, após a exibição da visão, executando os pontos de junção representados pela anotação *After*.

A anotação *AfterThrowing* possibilita executar um ponto de junção após o retorno de um erro. No PHP é possível criar uma função para manipular os erros gerados e utilizar a função *set_error_handler()* para chamar esta função criada para a manipulação dos erros.

O framework CodeIgniter disponibiliza uma função chamada *_exception_handler()*, que é utilizada para manipular os erros gerados pelo PHP. Através desta função é realizada a execução dos pontos de junção representados pela anotação *AfterThrowing*.

Desta forma é possível interceptar as chamadas ao controlador e executar os conjuntos de junções representados no aspecto, criando um suporte a interesses transversais no framework CodeIgniter.

4.2 Desenvolvimento da aplicação

Para analisar o projeto desenvolvido, foi criada uma aplicação de gerenciamento de cadastro de alunos, contendo um controle de acesso ao sistema e gerenciamento de alunos para cadastrar, atualizar e excluir um aluno do sistema.

Através desta aplicação, foi realizado o controle de sessão e a geração de log do sistema utilizando os conceitos básicos do framework CodeIgniter.

Após o desenvolvimento foi utilizando o suporte de interesses transversais desenvolvido neste projeto. Assim, foi possível verificar a viabilidade na utilização do suporte de interesses transversais do projeto comparado a utilização normal do framework.

O framework CodeIgniter contém um sistema de geração de logs nativo, onde alguns logs são gerados automaticamente pelo framework, mas é possível gerar logs personalizados através da chamada da função *log_message()*.

Para ativar a geração de log, foi realizada a alteração no arquivo *config.php* e ativado o armazenamento de todos os tipos de mensagem, onde por padrão o log é armazenado na pasta system/logs/.

Além das mensagens armazenadas pelo framework, foram adicionadas outras mensagens específicas da aplicação, assim, a cada chamada de um método do controlador foram armazenados os parâmetros enviados para o controlador e uma mensagem simples.

O controle de acesso ao sistema foi realizado a partir do suporte a sessão do próprio framework, armazenando o nome do usuário e a autenticação foi realizada com sucesso. Assim, ao realizar o acesso ao sistema, é armazenada a sessão do usuário permitindo acesso a área restrita da aplicação.

Para tela de acesso ao sistema, foram criados dos campos na visão *login_view.php*, solicitando o nome de usuário e senha para acesso. A Figura 21 representa a tela de acesso da aplicação.

Figura 21 - Tela de acesso da aplicação.

Fonte: Cláudio Rosse Pandolfi

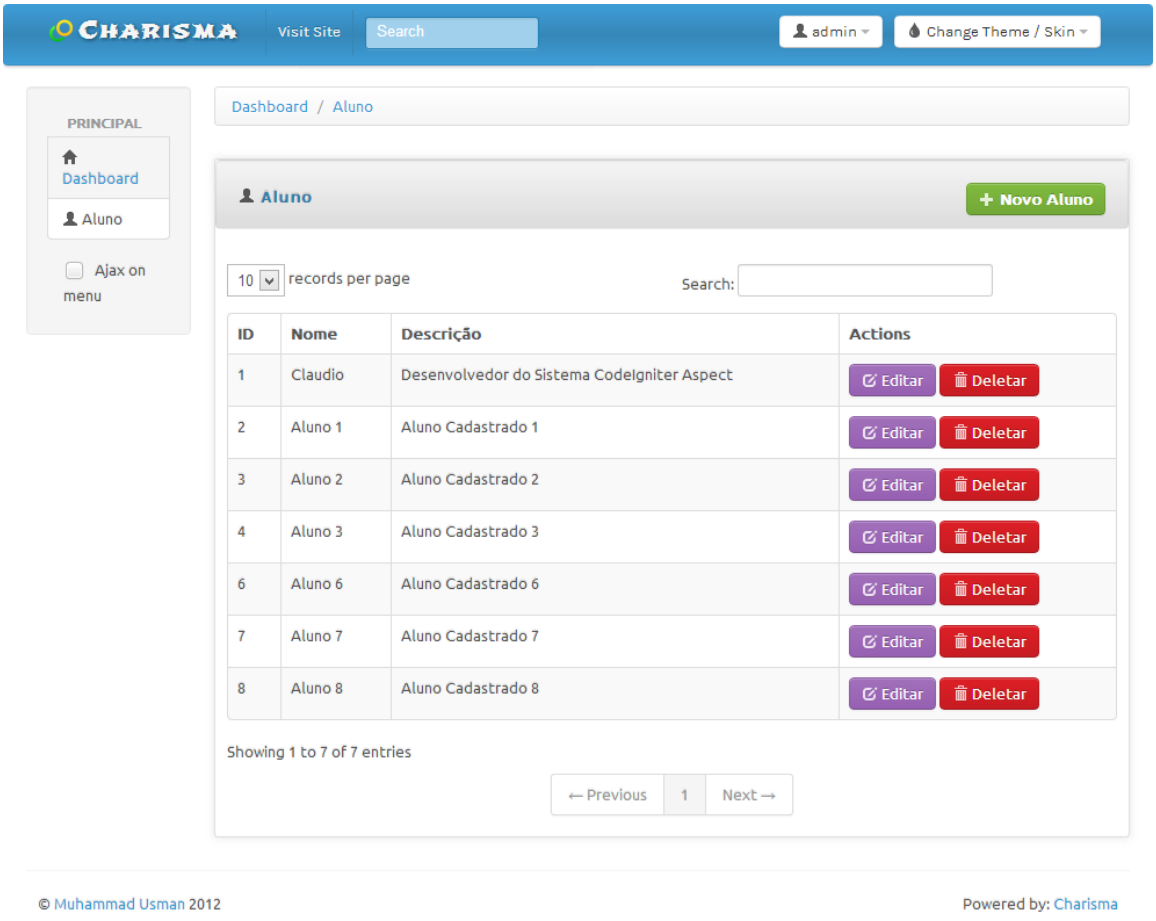
Foi criado o modelo *Membership_model* para validação do acesso ao sistema, através do método *validate()*, que verifica se o acesso ao sistema está correto, e o método *logged()*, que verifica a sessão armazenada e retorna se o usuário está autenticado.

No controlador *Login* foram realizadas as validações dos campos e a autenticação do usuário, a partir do método *index()*. Desta forma, caso atenda aos critérios de autenticação, uma sessão é armazenada e redirecionado o usuário para a área restrita da aplicação.

Para o armazenamento de log da aplicação, no método *index()* foi inserida algumas mensagens no início, que informam a tentativa de acesso ao sistema e os parâmetros enviados para a autenticação, e no final, que informam se a autenticação foi realizada com sucesso ou se houve algum problema ao realizar o acesso.

Ao realizar o acesso à aplicação, o usuário é redirecionado para a tela de lista de alunos, representada pela visão *alunos_view.php*, que contém todos os alunos cadastrados e as ações para inserir, editar os dados e apagar um aluno, conforme pode-se observar na Figura 22.

Figura 22 - Tela de Lista de Alunos.



Fonte: Cláudio Rosse Pandolfi

O modelo *Aluno_model* contém os métodos para a representação dos dados do aluno e, através dos métodos *select()*, *insert()*, *update()* e *delete()*, foi possível listar, inserir, editar e apagar um aluno do banco de dados.

Para a listagem dos alunos foi criado o controlador *Alunos*, que contém o método *index()*, que recupera os alunos cadastrados a partir do modelo *Aluno_model*. Assim, ao recuperar esta lista, envia as informações para a visão *alunos_view.php* para a exibição dos alunos cadastrados. Foram inseridas algumas mensagens de log do sistema, no início e no

final do método, para informar o acesso a tela de lista de alunos e se a exibição foi realizada com sucesso.

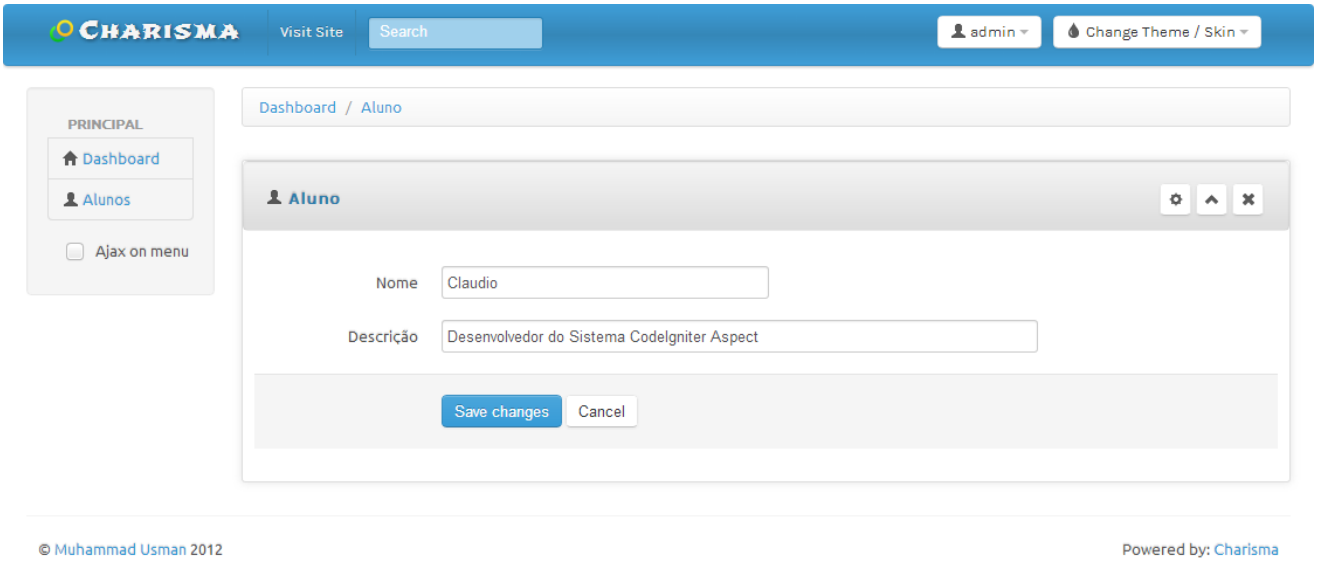
Através do método *__construct()* do controlador *Alunos*, foi realizado o controle de sessão, utilizando o método *logged()* do modelo *Membership_Model*. Caso o usuário não tenha uma sessão válida, o mesmo é redirecionado para a tela de acesso ao sistema.

No método *index()* foram inseridas duas mensagens para armazenar o log da aplicação, onde as mensagens representa o início ao acesso a lista de alunos e os dados retornados na lista.

Quando realizada alguma ação na tela de lista de alunos, é enviada uma requisição para o controlador *Aluno*, onde foi realizado todo o controle para cadastrar, editar e apagar um aluno na aplicação.

Ao realizar a ação para cadastrar um novo aluno, este controlador exibe a visão aluno_view.php, que contém os campos para cadastro de um novo aluno, conforme pode-se observar na Figura 23.

Figura 23 - Tela de Dados de Aluno.



Fonte: Cláudio Rosse Pandolfi

A visão *aluno_view.php* também foi utilizada para a edição de alunos já cadastrados, onde são recuperados os dados do aluno através do modelo *Aluno_model* e exibidos nos respectivos campos.

O controle de sessão também foi inserido dentro do método *__contruct()* da classe *Aluno*, realizando a chamada do método *logged()* do modelo *Membership_Model*, para permitir somente a edição dos dados através de um usuário que tenha acessado o sistema.

O log armazenado do controlador *Aluno* foi realizado através de mensagens em cada ação, onde foram registradas as chamadas dos métodos *insert(), update()* e *delete()*, os parâmetros enviados e se houve sucesso para execução da ação.

Com a aplicação desenvolvida a partir do framework CodeIgniter, foi realizado os ajustes para o controle de sessão e o armazenamento de logs a partir do suporte a interesses transversais desenvolvido neste projeto. Assim foram criados dois aspectos chamados *SessionAspect* e *LogAspect* para realizar o controle de sessão e armazenamento de log.

Para identificar que a classe criada representa um aspecto, foi incluso a anotação acima do nome da classe, indicando que a mesma representa um aspecto da aplicação, conforme ilustrado na Figura 24.

Figura 24 - Representação dos aspectos SessionAspect e LogAspect.

```
/**
* @Aspect
*/
class SessionAspect{
    ...
}

/**
* @Aspect
*/
class LogAspect{
    ...
}
```

Fonte: Cláudio Rosse Pandolfi

Através do aspecto *SessionAspect*, foi criado o método *sessionUser()* para o controle de sessão. Neste método foi inserido o código utilizado no método *logged()*, do modelo

Membership_Model, para verificar a sessão armazenada e retorna se o usuário está autenticado.

No método *sessionUser()*, foram adicionadas as anotações (Figura 25) para a interceptar todos os controladores e todos os métodos, exceto o controlador da página de acesso ao sistema. Assim, caso seja acessado qualquer página da aplicação, será executado este método antes de enviar a requisição para o controlador.

Figura 25 - Anotação do método sessionUser() do aspecto SessionAspect.

```
/**
* @Pointcut(execution='!Login *(*) || * *(*)')
* @Before
*/
function sessionUser() {
    ...
}
```

Fonte: Cláudio Rosse Pandolfi

Ao analisar a Figura 25, pode-se notar que a anotação *Pointcut* representa a interceptação de todos os controladores e métodos da aplicação, exceto os métodos do controlador *Login*, desta forma, caso seja acessado qualquer página da aplicação, será realizado o controle de sessão. Já o adendo *Before* indica que o aspecto será executado antes da execução de qualquer controlador.

Para o log da aplicação, foram realizados os ajustes para que o aspecto *LogAspect* fosse responsável por este armazenamento. A partir deste aspecto foram criados os métodos *logInitRequest()*, *logDataRequest()* e *logCompleteRequest()* para a interceptação dos controladores antes e após a execução da requisição, onde também o armazenamento dos dados da requisição enviada.

Nos métodos *logInitRequest()* e *logCompleteRequest()* foi adicionada a chamada da função *log_message()*, armazenando o log o início e fim da execução, registrando o controlador e método que foi executado na requisição.

Na Figura 26 pode-se observar as anotações adicionadas no método *logInitRequest()*, para a interceptação dos controladores.

Figura 26 - Anotação do método logInitRequest() do aspecto LogAspect.

```
/**
* @Pointcut(execution='* *(*)')
* @Before
*/
function logInitRequest() {
    ...
}
```

Fonte: Cláudio Rosse Pandolfi

Pode-se observar que a anotação *Pointcut* intercepta todos os controladores e métodos da aplicação, desta forma, caso seja acessado qualquer página da aplicação, é armazenado o controlador e método executado no log da aplicação. Para a interceptação deste aspecto antes da execução do controlador foi adicionada a anotação do adendo do *Before*.

Na Figura 27 pode-se observar as anotações inseridas no método *logCompleteRequest()*. Da mesma forma que na anotação do método *logInitRequest()*, o *Pointcut* intercepta todos os controladores e métodos da aplicação e armazena estas informações no log da aplicação, mas a diferença está no adendo *After*, que faz a interceptação do aspecto depois da execução do controlador.

Figura 27 - Anotação do método logCompleteRequest() do aspecto LogAspect.

```
/**
* @Pointcut(execution='* *(*)')
* @After
*/
function logCompleteRequest() {
    ...
}
```

Fonte: Cláudio Rosse Pandolfi

Para o armazenamento dos dados enviados na requisição, foi criado o método *logDataRequest()*, desta forma, foi possível armazenar, no log da aplicação, os dados enviados em algumas requisições.

No método *logDataRequest()* foram adicionadas as anotações para interceptar os dados enviados na requisição ao ser realizado o acesso ao sistema e quando realizada alguma ação para cadastrar, editar e apagar um aluno, conforme figura 28.

Figura 28 - Anotação do método logDataRequest() do aspecto LogAspect.

```
/**
* @Pointcut(execution='Login index(*) || Aluno insert(*)
|| Aluno update(*) || Aluno delete()')
* @AfterReturning @AfterThrowing
*/
function logDataRequest() {
    ...
}
```

Fonte: Cláudio Rosse Pandolfi

A execução desse aspecto é realizada logo após o retorno do controlador, definido através da anotação *AfterReturning*, e caso ocorra algum problema na execução da aplicação, foi inserida a anotação do adendo *AfterThrowing*, registrando os dados da requisição.

4.3 Analise sobre a utilização do Suporte a Interesses Transversais

Para a análise do projeto desenvolvido, foram realizadas algumas comparações entre o desenvolvimento das duas aplicações, uma utilizando o framework CodeIgniter e outra utilizando o suporte a interesses transversais desenvolvido neste projeto.

Ao realizar o controle de sessão pelo framework CodeIgniter, foi criado o método *logged()*, no modelo *Membership_model*, para verificar a sessão armazenada e retornar se o usuário estava autenticado. Assim, foi adicionada a chamada deste método no construtor dos controladores para verificar se o usuário estava autenticado, o que acarretou a chamada do método *logged()* em diversos pontos da aplicação.

Ao utilizar o aspecto para o controle de sessão, pode-se notar que não foi necessário incluir uma chamada no construtor do controlador, devido ao conjunto de junções adicionados no método *sessionUser()* do aspecto *SessionAspect*, e caso seja criado um novo controlador na aplicação, por exemplo, o controle para as notas do aluno, o aspecto irá interceptar este novo controlador.

Neste aspecto pode-se considerar também a fácil manutenção, pois caso seja necessário realizar algum ajuste no controle de sessão, basta realizar os ajustes no aspecto, onde este ajuste afetará toda a aplicação.

Podem existir algumas páginas da aplicação que não terão a necessidade do controle de sessão, sendo assim, para que não seja realizado o controle de sessão através do aspecto *SessionAspect*, basta adicionar uma exceção no conjunto de junção, informando o controlador ou métodos que não serão necessários neste controle de sessão.

O armazenamento do log da aplicação através do *LogAspect* também apresenta a vantagem de manutenção da aplicação em relação ao armazenamento realizado utilizando somente o framework CodeIgniter.

Através do framework CodeIgniter, foi necessário adicionar a chamada da função *log_message()* em diversos pontos da aplicação para o armazenamento do log, onde as execuções dessa função foram realizadas diretamente pelo controlador. Assim, a chamada desta função pelo controlador diminui a modularização da aplicação, visto que houve uma dependência desta chamada durante a execução do mesmo.

Com o aspecto *LogAspect* não foi necessário realizar as chamadas da função *log_message()* em diversos pontos da aplicação, onde a chamada desta função foi realizada nos métodos *logInitRequest(), logDataRequest()* e *logCompleteRequest()* do aspecto, que realizam a interceptação da execução do controlador e armazena o log conforme a anotação adicionada em cada método.

Ao modularizar este log da aplicação, utilizando o aspecto, foi possível padronizar as mensagens armazenadas, o que facilitou na interpretação dos registros, e também possibilitou que outro desenvolvedor poderia ajustar a aplicação, visto que o aspecto é desenvolvido independente da aplicação.

Devido a expressão adicionada nos métodos *logInitRequest()* e *logCompleteRequest()*, caso seja adicionado um novo controlador na aplicação, já será armazenado o log ao acessar este novo controlador.

No método *logDataRequest()* foram adicionadas anotações especificas para interceptar os dados enviados na requisição para cadastrar, alterar ou excluir alunos da aplicação, onde também foram adicionadas anotações na tentativa de acessa na aplicação.

Como no desenvolvimento de software muitas vezes são utilizadas nomenclaturas padrões nos métodos, onde muitas vezes são encontrados métodos com nomes idênticos em classes distintas.

Desta forma, se ao criar um novo controlador, por exemplo para o cadastro das notas dos alunos, e neste controlador adicionados os métodos *insert()*, *update()* e *delete()*, bastaria somente realizar o ajuste na anotação do aspecto para interceptar estes métodos, independente do controlador existente na aplicação, conforme pode-se observar na Figura 29.

Figura 29 - Ajuste da anotação do método logDataRequest().

```
/**
* @Pointcut(execution='Login index(*) || * insert(*)
|| * update(*) || * delete()')
* @AfterReturning @AfterThrowing
*/
function logDataRequest() {
    ...
}
```

Fonte: Cláudio Rosse Pandolfi

Ao se utilizar o caractere asterisco no conjunto de junções conforme Figura 29, foi generalizado o comportamento do aspecto, assim independente do controlador executado, caso seja chamado algum método com a nomenclatura *insert*, *update* ou *delete*, o método *logDataRequest* será executado.

CONSIDERAÇÕES FINAIS

A utilização do suporte a interesses transversais mantém maior modularização da aplicação, visto que pode-se trabalhar com os interesses transversais sem afetar a aplicação desenvolvida.

Através do projeto desenvolvido foi possível separar os interesses transversais e modularizar a aplicação, utilizando os conceitos de programação orientada a aspectos, disponibilizando ao desenvolvedor um conjunto de técnicas para esta nova abstração através de mecanismos de composição de aspectos e componentes.

Pode-se notar que o desenvolvimento dos aspectos SessionAspect e LogAspect são independentes da aplicação, o que possibilitaria a outros desenvolvedor realizar ajustes na aplicação, enquanto outros desenvolvedores realizam ajustes nestes aspectos.

Através do aspecto SessionAspect foi possível centralizar o controle de sessão do usuário, o que facilitou o desenvolvimento, visto que não houve a necessidade de inserir este controle de sessão em todo método construtor dos controladores criados na aplicação.

O aspecto LogAspect possibilitou, além da facilidade de manutenção devido a centralização do código, evitar que a chamada do armazenamento de log em diversas partes da aplicação, trazendo benefícios no desenvolvimento visto que não foi necessário realizar ajustes em toda a aplicação para armazenar novas informações, principalmente na criação de novos controladores.

Desta forma, foi possível disponibilizar esta nova abstração para os requistos não funcionais da aplicação, onde desenvolvedores podem se dedicar em necessidades da aplicação, enquanto outro desenvolvedor pode se dedicar na implementação desses aspectos para compor os interesses transversais da aplicação.

REFERÊNCIAS

ALVES. Júlio Cesar, ALVES. Flávio Luis, ROCHA. Anderson de Rezende, SOARES. Alexandre Henrique Vieir, **Programação Orientada a Aspectos – Uma Visão Geral**, Departamento de Ciência da Computação, Universidade Federal de Lavras.

ARAUJO, Alessandro Cruvinel Machado de, MENESES, Javé Barbosa de, **Requisitos Não Funcionais**, Departamento de Informática, Universidade Federal de Pernambuco, 2000.

BALTHAZAR, Glauber da Rocha, GUIMARÃES, Fabio Mendes Ramos, PAULA, Melise Maria Veiga de, FILHO, Elio Lovisi, **Uma Abordagem Prática sobre a Aplicação do Padrão MVC com o Framework Struts**, Faculdade Metodista Granbery, 2006.

CAMPOS, Marcelo Ricardo, **Compreensão Visual de Frameworks através de Introspeção de Exemplos**, Instituto de Informática, Universidade Federal do Rio Grande do Sul, 1997

CARMO, Carla Soraia Leandro do, **Automação de detalhamento de peças padronizadas em concreto armado via CAD e programação orientada a objetos**, Departamento de Engenharia de Estruturas, Universidade Federal de Minas Gerais, 2001.

CYSNEIROS, Luiz Marcio, **Requisitos Não Funcionais: Da Elicitação ao Modelo Conceitual**, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 2001.

CodeIgniter / EllisLab, em Disponível em: <http://ellislab.com/codeigniter>, acessado em 19 de Nov. de 2013.

CodeIgniter User Guide Version 2.1.4, em Disponível em: <http://ellislab.com/codeigniter/user-guide/>, acessado em 19 de Nov. de 2013.

FARINELLI, Fernanda, Conceitos **Básicos de Programação Orientada a Objetos**, Instituto Federal Sudeste de Minas Gerais 2007.

GABARDO, Ademir Cristiano, **PHP e MVC com Codeigniter**, Editora Novatec, 2012.

GONÇALVES, Edson, **Desenvolvendo Aplicações Web com JSP Servlet, JavaServer Faces, Hibernate, EJB 3 Persistence e Ajax**, Editora Ciência Moderna Ltda., 2007.

HUGO. Marcel, **Estudo de Caso Aplicando Programação Orientada a Aspecto**, Departamento de Sistemas e Computação, Universidade de Blumenau, 2005.

JOHNSON, Rod, **How to Design Frameworks, em Conference on Object-Oriented Programming, Languages and Applications**, 1993.

JOHNSON. Ralph E., **Components, Frameworks, Patterns**, 1997, White paper.
KICZALES, Gregor, LAMPING John, MENDHEKAR , Anurag, MAEDA, Chris, LOPES, Cristina Videira, LOINGTIER, Jean-Marc, and John Irwin. **Aspect-Oriented Programming, em European Confer- ence on Object-Oriented Programming**, 1997.

LASKOSKI, Jackson, **Programação Orientada a Objetos**, apostila, Disponível em: < http://www.jack.eti.br/**>.** Acesso em 19 de Nov. de 2013.

LOBATO, Cidiane Aracaty, **Um Framework Orientado a Aspectos para Mobilidade de Agentes de Software**, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 2005.

LOUREIRO, João Manuel Bonita Pereira, COSTA, João Pedro Couto Soares Gonçalves da, FONSECA, Rossana Mendes Sequeira Baptista da, NEVES, Vergílio Augusto. **Programação Orientada a Aspectos**, Faculdade de Engenharia, Universidade do Porto, 2005.

MACORATTI, José Carlos. **Padrões de Projeto : O modelo MVC - Model View Controller**, Disponível em: <http://www.macoratti.net/vbn_mvc.htm> .Acesso em 01 de Out. de 2013.

MARIANI. Antonio Carlos, **O Mundo dos Atores: uma perspectiva de introdução à programação orientada a objetos**, Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, 1999.

MEYER, Bertrand. **Object-Oriented Software Construction**. Editora Prentice-Hall 1997.

MINETTO, Elton Luís, Framework **para Desenvolvimento em PHP**, Editora Novatec, 2007.

OSSHER, Harrison, **Specifying suject–oriented composition**, Special Issue on Subjectivity in OO Systems. 1996.

OSSHER Harold, TARR Peri. **Using subject–oriented programming to overcome common problems in object–oriented software development/evolution**. em International Conference on Software Engineering, 1999.

RICARTE, Ivan Luiz Marques, **Programação Orientada a Objetos: Uma Abordagem com Java**, Departamento de engenharia de computação e automação industrial, Universidade Estatual de Campinas 2001.

RODRIGUES, Francisco Aparecido, **Técnicas de orientação ao objetos para computação científica paralela**, Departamento de Física e Informática, Instituto de Física de São Carlos, Universidade de São Paulo, 2004.

SILVA, Elaine Quinteiro da, MOREIRA, Dilvan de Abreu, **Um Framework de Componentes para o Desenvolvimento de Aplicações Web Robustas de Apoio à Educação**, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2004.

SILVA, Lyrene Fernandes da, **Uma Estratégia Orientada a Apectos para Modelagem de Requisitos**, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, 2006.

SILVA. Jaguaraci Batista, BARRETO. Luciano Porto, **Separação e Validação de Regras de Negócio MDA através de Ontologias e Orientação à Aspectos**, Departamento de Ciência da Computação, Universidade Federal da Bahia, 2008.

SILVA. Ricardo Pereira, **Suporte ao Desenvolvimento e uso de Frameworks e Componentes**, Instituto de Informática, Universidade Federal do Rio Grande do Sul, 2000.

SOARES, Sérgio Castelo Branco. **An Aspect-Oriented Implementation Method**. Centro de Informática, Universidade Federal de Pernambuco, 2004.

SOARES. Ségio, BORBA. Paulo, **Desenvolvimento de Software Orientados a Aspectos Utilizando RUP e AspectJ**, Centro de Informática, Universidade Federal de Pernambuco, 2004.

TALIGENT, Leveraging. **Object-Oriented Frameworks**, White Paper, 1995.

The NetBeans E-commerce Tutorial - **Designing the Application**, Disponível em: <https://netbeans.org/kb/docs/javaee/ecommerce/design.html>. Acessado em 31 de Maio de 2013.