CENTRO UNIVERSITÁRIO DINÂMICA DAS CARATATAS

CURSO DE SISTEMAS DE INFORMAÇÃO

COMPARATIVO ENTRE PLAY FRAMEWORK E RUBY ON RAILS

HAROLDO RAMIREZ DA NÓBREGA

FOZ DO IGUAÇU - PR

2015

HAROLDO RAMIREZ DA NÓBREGA

COMPARATIVO ENTRE PLAY FRAMEWORK E RUBY ON RAILS

Monografia de conclusão de Curso como requisito parcial para a obtenção do grau de Bacharel em Sistemas de Informação do Centro Universitário Dinâmica das Cataratas – UDC, sob a orientação do Prof. Fabiano Damin.

FOZ DO IGUAÇU - PR

2015

TERMO DE APROVAÇÃO

HAROLDO RAMIREZ DA NÓBREGA

COMPARATIVO ENTRE PLAY FRAMEWORK E RUBY ON RAILS

Trabalho de Conclusão de Curso apresentado ao Centro Universitário Dinâmica das Cataratas, curso de Sistemas de Informação para a obtenção do grau de Bacharel em Sistemas de Informação, aprovado pela banca examinadora formada por:

_____

Orientador: Profº Fabiano Damin

BANCA EXAMINADORA

_____

Profº Adélio Conter

_____

Profº. Msc. Sérgio Augusto S. Lopes

Foz do Iguaçu, junho de 2015.

TERMO DE COMPROMISSO DE ORIENTAÇÃO

Eu, _____, que a esta subscrevo, assumo o compromisso de cumprir o disposto no artigo 17 do Regulamento 002/99, bem como as demais disposições do mesmo, bem como as regras que venham a alterá-lo ou completá-lo, como o objetivo de realizar o trabalho monográfico de conclusão de curso exigido pelo artigo 9º da Portaria 1886/94 com MEC.

Foz do Iguaçu, _____

TERMO DE COMPROMISSO DE ORIENTAÇÃO

Eu, _____, professor da disciplina de

_____, que a esta subscreve, assumo o compromisso de cumprir o disposto no artigo 14 do regulamento 002/99, bem como as demais disposições do mesmo, como a outras que venham a alterá-lo ou completa-lo, com o objetivo de orientar a realização do trabalho monográfico de conclusão de curso exigido pelo artigo 9º da Portaria 1886/94 com MEC.

Foz do Iguaçu, _____

Dedico este trabalho a toda minha família e amigos, e a todos que sempre permaneceram realmente comigo durante estes períodos da graduação, e que ajudaram e me fortaleceram cada um de uma forma. E a todos os meus professores aos quais abriram minha mente para adquirir o rico conhecimento necessário para o desenvolvimento.

AGRADECIMENTOS

Agradeço a minha namorada Bruna Natasha Rial Rosa por sempre estar ao meu lado e pela grande ajuda na decisão da escolha do tema deste trabalho.

Meus agradecimentos a os pais Haroldo Italiano da Nóbrega e Adela Fidela Ramirez da Nóbrega por sempre acreditarem em mim, não medindo esforços para me ajudar a conquistar meus objetivos.

Aos meus colegas de sala agradeço pelo companheirismo e união no decorrer destes anos.

Ao meu professor e orientador Fabiano Damin por ter acreditado no meu potencial e me orientado neste semestre para a elaboração deste trabalho.

Ao professor Adélio Conter pela visão ampla de conhecimento e as cobranças para concluir o curso.

Também agradeço ao professor Jorge Aikes pela ajuda na definição do tema e ter me apresentado o *framework* chamado Play Framework.

Agradeço também ao Instituto de Tecnologia Aplicada e Inovação ITAI por me dar esta valiosa oportunidade de estágio e o tempo necessário para a realização deste trabalho.

Ao Fernando Geraldo Mantoan e Anderson Davi pelo suporte e incentivo para a utilização do Play Framework.

“Um *framework* não é bom pelo que ele faz, mas quando pode ser utilizado para o que ele não faz.”

Eduardo Guerra

“Penso noventa e nove vezes e nada descubro; deixo de pensar, mergulho em um profundo silêncio – e eis que a verdade se me revela.”

Albert Einstein

RESUMO

Atualmente as empresas de desenvolvimento buscam ferramentas para agilizar na criação de aplicativos de qualquer natureza, sendo ela Web, Desktop ou Mobile. Essas ferramentas são chamadas de *Frameworks*, cada uma dessas soluções tem várias funções para o uso no desenvolvimento, evitando assim os erros e melhorando a qualidade final do *software*. Existem várias ferramentas no mercado, sendo que o grande desafio é escolher a mais adequada para determinada situação. A proposta deste trabalho é ajudar o programador na tomada de decisão entre qual *framework* escolher, Play Framework ou Ruby on Rails entre as várias outras no mercado como Spring MVC, Django e Jersey.

Palavras-Chave: Web, Framework, Arquitetura de desenvolvimento, RESTFul, *Design Patterns*.

ABSTRACT

Today the development companies are looking for tools to speed the creation of any type of application, it being Web, Desktop or Mobile. These tools are called frameworks, each of these solutions has various functions for use in the development, thus avoiding mistakes and improving the final quality of the software. There are several tools on the market but the big challenge is to choose the most appropriate for a given situation. The purpose of this work is to help the programmer in decision making framework among which to choose, Play Framework or Ruby on Rails among several others on the market as Spring MVC, Django and Jersey.

Keywords: *Web, Framework, Development arquiteture, RESTFul, Design Patterns.*

LISTA DE FIGURAS E TABELAS

LISTA DE SIGLAS

GoF……Gang of Four

HTML……HyperText Markup Language

HTTP…….....HyperText Transfer Protocol

IDE.....Integrated Development Environment

MVC…….....Model View Controller

TDD……. .....Test Driven Development

SQL……...…Structured Query Language

SUMÁRIO

1. INTRODUÇÃO

A informática e o uso dos computadores estão oferecendo melhorias em quase todas as áreas de trabalho. Cada dia mais pessoas interagem com *softwares* e a internet também faz a sua parte espalhando informações a cada segundo. Cada ano o computador sofre modificações na engenharia tornando o mais rápido, menor e mais barato de se fabricar.

Atualmente no mercado existe uma grande quantidade de ferramentas chamadas de *frameworks*, que são soluções para desenvolver *softwares* em uma pequena quantidade de tempo, as empresas que desenvolvem aplicativos, tem uma busca constante para criar sistemas de maneira ágil, fácil e prática. Os *frameworks* têm suas próprias bibliotecas e soluções prontas para serem exportadas no projeto e usar como o programador desejar. Essas bibliotecas são como um conjunto de funções que ajudam a desenvolver soluções de forma rápida, pois não é necessário criar essas funções novamente, mas sim apenas usá-las.

Existem várias maneiras de se desenvolver *softwares* e os projetistas buscam criar programas cada vez melhores e mais complexos, para isso existem muitas etapas que devem ser seguidas rigorosamente para que assim se consiga concluir um projeto maduro com o mínimo de falhas possíveis.

Todas as nações são dependentes de programas complexos, *software* de computadores, assim como a infraestrutura e os serviços nacionais são dependentes, a maioria dos aparelhos elétricos vem incluso um *software* embarcado que controla suas funções (SOMMERVILLE, 2007).

1.1. OBJETIVOS

1.1.1. OBJETIVOS GERAIS

O objetivo deste trabalho consiste em realizar um comparativo entre os frameworks, Play Framework e Ruby on Rails, apoiando a tomada de decisão no desenvolvimento de sistemas Web.

1.1.2. OBJETIVOS ESPECÍFICOS

- Fazer uma abordagem sobre os padrões de projetos de *software*;
- Apresentar as arquiteturas de *software* e as suas respectivas camadas físicas;
- Expor os *frameworks* de desenvolvimento;
- Descrever os dois *frameworks* existentes no mercado sendo eles Play framework e Ruby on Rails;
- Discutir sobre as metodologias de comparação de *frameworks*;
- Realizar o comparativo entre o Play framework e o Ruby on Rails.

1.2. JUSTIFICATIVA

Este trabalho se justifica na questão de comparar os *frameworks* Play Framework e Ruby on Rails, para permitir que desenvolvedores tenham informações suficientes para a tomada de decisão de qual destes devem utilizar, dado os desafios que iram enfrentar durante o desenvolvimento de soluções Web e *Mobile*.

Existem várias opções de *frameworks*, tornando a escolha do melhor uma tarefa difícil. Este trabalho vai ajudar o programador nessa decisão, pois irá comparar um *framework* com mais tempo no mercado sendo o Ruby on Rails e a proposta do mais atual que é o Play Framework.

1.3. PROBLEMA

Como definir um critério de comparação entre os *frameworks* Play Framework e Ruby on Rails com o objetivo de apoiar a tomada de decisão?

1.4. METODOLOGIA

Para o desenvolvimento do trabalho, a primeira etapa considerada foi a busca do material necessário para pesquisa, realizada por meio de um levantamento bibliográfico, esse levantamento foi feito através de pesquisas em livros e fontes da internet,

foi coletado o maior número de fontes e recursos disponíveis para a melhor qualidade e desenvolvimento da produção desta Monografia.

Após as considerações do material bibliográfico escolhido, a próxima etapa expos o tema dos padrões de projetos que é a base para a criação de soluções de *softwares,* sendo que a adoção dos vários padrões existentes no mercado possam auxiliar para muitos aspectos do *software*, tornando o fácil de manter, com menor quantidade de erros e agilidade no desenvolvimento.

A próxima etapa contempla a explicação de como são e funcionam as arquiteturas de *softwares* existentes no mercado, expondo as camadas de *software*, pois atualmente considera-se que para melhor modularidade e escalabilidade é necessário dividir o sistema em várias partes.

Na sequência serão explicados fundamentos sobre os *frameworks* de desenvolvimento WEB e serão apresentados os *frameworks* chamados de Play Framework e Ruby on Rails. Após a apresentação dos *frameworks* serão realizadas pesquisas sobre as metodologias de comparação destes *frameworks*.

As últimas etapas da pesquisa contemplam a definição dos critérios de comparação e a realizam baseada nos critérios já definidos anteriormente.

Com essa comparação realizada o próximo passo será a demonstração de resultados baseados na comparação realizada sobre os critérios estabelecidos e a conclusão do trabalho desenvolvido.

Desta forma os procedimentos metodológicos a serem desenvolvidos durante a execução deste trabalho dão a importância devida ao seu conteúdo para a sua conclusão com qualidade e desenvoltura.

1.5. ESTRUTURA DO TRABALHO

O capítulo I traz a introdução, cuja temática se baseia nos objetivos, especificamente o objetivo geral e específico. A justificativa do trabalho segue complementada pelo problema a ser resolvido através dos tópicos correlacionados nas páginas a seguir.

O capítulo II faz uma abordagem sobre os padrões de projetos, conhecido também como *Design Patterns* do grupo *Gang of Four* (GOF).

O capítulo III introduz informações sobre as arquiteturas de *software* expondo

a arquitetura monolítica, duas camadas, três camadas e n. camadas.

O capítulo IV discorre sobre os *frameworks* de desenvolvimento, comentando um pouco sobre os *frameworks* Play Framework e o Ruby on Rails.

O capítulo V comenta sobre as metodologias de comparação de *frameworks*, fazendo uma pesquisa das comparações existentes na literatura.

O capítulo VI mostra um comparativo entre os *frameworks* Play Framework e Ruby on Rails. Expondo e explicando seus pontos fortes e fracos.

O capítulo VII conclui o tema de forma descritiva e objetiva.

2. PADRÕES DE PROJETOS

Para fazer com que os programadores pensem da mesma maneira impedindo que sejam causadas confusões na hora de escolher a melhor solução para o negócio FREEMAN et al. (2009), defende que quando os padrões de projetos são adotados de forma correta, podem facilitar o desenvolvimento de sistemas.

Como afirma GUERRA (2013), padrões de projetos é um conjunto composto por um problema, uma solução dentro de um contexto, sendo que essa solução tenha sido utilizada com sucesso em mais de um contexto e que o padrão não relata novas soluções, mas sim soluções sólidas.

Os padrões de projetos não são ideais originais, para FOWLER (2006) eles não foram inventados e sim descobertos através de observações que ocorrem na prática, sendo que a sua função é de comunicar essa ideia através de nomes assim facilitando a comunicação entre os projetistas de *software*.

Segundo GAMMA et al. (2000), embora o engenheiro civil chamado Christopher Alexander estivesse falando sobre os padrões em construções de cidades, a sua afirmação também é verdadeira em relação aos padrões de projetos orientado a objetos. Os padrões de projetos facilitam na hora de escolher alternativas para tornar o sistema reutilizável e evitar alternativas que comprometa a sua reusabilidade, podendo melhorar a manutenção e a documentação de sistemas já existentes.

Enquanto que, para HORTMANN (2007), cada *design pattern,* chamados de padrões de projeto é descrito de um problema e uma solução, que pode ser aplicada em várias situações de programação.

Os padrões de projeto podem ser classificados a partir de seu propósito que, segundo GAMMA et al. (2000), pode ser de comportamento, de criação ou de estruturação:

• Padrões de criação são focados no processo de criação de objetos;

• Padrões de estruturação cuidam da composição de classes ou de objetos;

• Padrões de comportamento caracterizam-se pelos meios em que as classes ou os objetos irão interagir e distribuir suas responsabilidades.

As soluções propostas pelos padrões de projetos são para problemas que

acontecem no desenvolvimento de *software* e que essas soluções possam ser reutilizadas por outros desenvolvedores.

2.1. CLASSIFICAÇÃO DOS PADRÕES DE PROJETOS

A figura 01 demonstra as divisões do escopo que são as classes e os objetos com os propósitos de criação, estrutura e comportamento, presente nos padrões de projeto, isso ajuda na hora de entender cada padrão e qual sua finalidade.

		Propósito		
		Criação	Estrutura	Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Builder Abstract Factory Prototype Singlenton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Figura 01 – O espaço dos padrões de projeto
Fonte: Baseado em GAMMA et al. (2000)

Segundo GAMMA et al. (2000), o critério chamado escopo mostra se o padrão é aplicável a classes ou aos objetos, os padrões de classes lidam com classes e subclasses através de seus relacionamentos, sendo que esses relacionamentos são através da herança. Os padrões de objetos são mais dinâmicos, pois os relacionamentos entre os objetos podem mudar em tempo de execução.

Padrões de criação têm como objetivo abstrair o processo de criação de objetos, ou seja, a sua instanciação. Fazem com que o sistema seja independente de como são criados e representados os seus objetos. O padrão de objetos delegará a sua instancia para outros objetos enquanto que o padrão de criação de classes faz o uso da herança para variar a classe que é instanciada (GAMMA et al., 2000).

Segundo GAMMA et al. (2000), os padrões classificados como de estruturas tratam como as classes e objetos são compostos para criar alicerces maiores. Este padrão tem duas categorias, sendo:

- Padrões estruturais de classe: usam herança para formar interfaces ou implementações.
- Padrões estruturais de objeto representam caminhos para criar objetos e atingir a novas funcionalidades.

Os padrões de comportamento que se referem às classes utilizam da herança para representar algoritmos e controle de fluxos, enquanto que o de objetos descreve como um determinado grupo de objeto vai trabalhar na forma de cooperação para realizar uma função que um único objeto é incapaz de realizar sozinho (GAMMA et al., 2000).

Os padrões de projetos que estão localizados no propósito de criação são responsáveis pela instanciação de objetos, eles administram como será criado cada objeto do sistema, no propósito de estrutura os padrões descrevem maneiras de como agrupar e organizar as classes, no propósito de comportamento os padrões envolvidos se responsabilizam em definir como cada objeto vai se comportar, delegando responsabilidades a algoritmos classes e objetos.

2.1.1. Padrões de Criação

2.1.1.1. Factory Method

Segundo HORSTMANN (2007), a proposta desse padrão é ajudar a atribuir responsabilidade para a criação de objetos. O *Factory Method* permite a uma classe delegar a instanciação para as suas subclasses. Fornece um método podendo ser sobrescrito para instanciar objetos de vários tipos.

Conforme GAMMA et al. (2000), o padrão *Factory Method* para instanciar um objeto é definida uma interface mas deixando que as subclasses decidirem qual classe instanciar permitindo assim adiar a instanciação para as subclasses. Isto remove a necessidade de anexar classes específicas das aplicações no código fonte.

2.1.1.2. Builder

De acordo com GAMMA et al. (2000), o objetivo do padrão *Builder* é separar a construção de um objeto complexo da sua representação, para que o mesmo processo de construção possa criar representações diferentes. Muito útil em construção de algoritmos complexos.

O padrão *Builder* oferece uma solução para a criação de objetos complexos, definindo uma classe responsável pelo processo de instanciação. A partir dessa criação é possível manter o mesmo processo de instanciar uma classe para diferentes representações e diferentes estruturas. Com isto a classe que chama os métodos de um *Builder* se mantem desacoplada da lógica complexa de criação (GUERRA, 2013).

2.1.1.3. Abstract Factory

A intenção o padrão *Abstract Factory*, segundo GAMMA et al. (2000), é fornecer uma interface para criar famílias de objetos relacionados ou dependentes, sem especificar suas classes concretas. Ao lidar com interfaces do usuário o sistema pode precisar usar um conjunto de objetos para trabalhar em um determinado sistema operacional e outra para um sistema operacional diferente.

Para GUERRA (2013), o objetivo do padrão *Abstract Factory* é de criar uma família de objetos que se relacionam entre si. Fazendo com que não exista inconsistência entre esses objetos, pois eles pertencem à mesma fábrica.

2.1.1.4. Prototype

Como afirma GAMMA et al. (2000), o objetivo do padrão *Prototype* é descrever os tipos de objetos fazendo com que esses próprios objetos se instanciem como protótipo, criando objetos apenas realizando uma cópia deles mesmos.

O padrão *Prototype* possibilita ao cliente instanciar novos objetos através da cópia de um exemplo. A diferença entre chamar um *constructor* e copiar um objeto é que o objeto copiado tem algo do estado do objeto original (METSKER, 2004).

2.1.1.5. Singleton

Segundo METSKER (2004), o padrão *Singleton* garante que uma classe tenha apenas uma única instância e forneca um ponto de acesso global a ela. A intenção do padrão sugere que um objeto específico carrega uma responsabilidade da qual outros objetos dependem.

Para SHALLOWAY E TROTT (2004), este padrão de projeto propõe de certificar que a classe tenha somente uma instância, fazendo com que tenha um acesso global a ela mesma.

2.1.2. Padrões Estruturais

2.1.2.1. Adapter

O padrão *Adapter* para GAMMA et al. (2000), é uma classe existente que executa os serviços que o cliente precisa, mas com a interface diferente do desejado, podemos aplicar o padrão *Adapter*, pois a intenção é fornecer a interface desejada, usando os serviços de uma classe existente com uma interface apropriada sem adaptações.

Para GUERRA (2013), o *Adapter* é semelhante a um adaptador de tomadas, onde criamos uma classe com a interface necessária, mas internamente a implementação é referente a interface antiga.

2.1.2.2. Bridge

Segundo GAMMA et al. (2000), o objetivo de padrão *Brigde,* é desacoplar uma abstração da implementação de suas operações abstratas, de modo que a abstração e sua implementação possam variar de forma independente. O uso comum ocorre quando uma aplicação utiliza um driver que é um objeto que opera no sistema operacional ou um dispositivo externo segundo uma interface bem especificada.

O padrão *Bridge* cria uma ponte sobre duas hierarquias ligadas por uma relação

de composição, permitindo que haja variações entre ambas de forma independente (GUERRA, 2013).

2.1.2.3. Composite

Composite é um conjunto de objetos no qual alguns deles podem conter outros objetos em estrutura de árvores para assim representar hierarquias, pode permitir aos clientes tratar objetos individuais e composições de objetos uniformemente. O primeiro conceito consiste em projetar grupos para que possam conter itens individuais ou outros grupos, o segundo é definir comportamentos comuns para objetos individuais e para composições (GAMMA et al., 2000).

O *Composite* tem o objetivo de permitir que a mesma abstração possa ser usada para uma instancia simples ou para o seu próprio conjunto, (GUERRA, 2013), afirma que essa abstração pode ser uma interface básica que precisa ser executada nos objetos simples e nos compostos.

2.1.2.4. Decorator

Para FREEMAN et al. (2009), o padrão *Decorator* permite compor o comportamento de um objeto dinamicamente. Estruturalmente o padrão *Decorator,* dispõe de classes em uma hierarquia e distribui operações ao longo dela. Cada classe dessa hierarquia tipicamente tem um construtor que precisa de outra instância de uma classe dela.

O padrão de projeto *Decorator* possibilita o acréscimo de funcionalidades a um objeto, e não a uma classe, em tempo de execução, provendo uma alternativa bem flexível como mecanismo de extensão de classes. O acréscimo de tais funcionalidades é realizado pelo encadeamento de um conjunto de objetos, sendo que os primeiros tratam das novas funcionalidades e o último da funcionalidade original (SHALLOWAY, 2004).

2.1.2.5. Facade

O padrão *Facade* é uma classe cuja suas funcionalidades são semelhantes a um *toolkit* que para METSKER (2004), fornecem uma interface que o torna um subsistema de fácil uso. Um *toolkit* é como um organizador de subsistemas que elabora pacotes com classes projetadas, mas sem propor nada que crie acoplamento.

Para GUERRA (2013), a palavra é de origem francesa, sua pronuncia é “façade”. Este padrão possibilita a criação de uma fachada, uma classe intermediária para que o cliente consiga ter acesso as funções desejadas gerando assim um encapsulamento da complexidade da relação entre os outros componentes fazendo com que haja um desacoplamento fraco entre o cliente e a execução.

2.1.2.6. Flyweight

Como explica METSKER (2004), o padrão *Flyweight* tem como objetivo aplicar o compartilhamento para suportar de forma eficiente um número realmente elevado de objetos muito pequenos. Em qualquer aplicação em que haja um grande número de pequenos objetos, podemos precisar fornecer uma maneira de os clientes compartilharem seguramente os elementos comuns desses objetos sem que haja um grande consumo de processamento.

A finalidade do padrão *Flyweight* é possibilitar a representação de um grande número de objetos da forma mais apropriada possível. Segundo GUERRA (2013), se aplica em situações onde existem várias instâncias da mesma classe na memória, sendo elas equivalentes. Para que isso ocorra, o padrão faz a reutilização da mesma instância em todos os locais onde os objetos parecidos precisem ser usados.

2.1.2.7. Proxy

Um objeto comum faz o seu próprio trabalho em suporte à interface pública que promove, mas pode acontecer que um objeto legítimo não consiga arcar com sua própria responsabilidade comum, isso pode ocorrer quando o objeto demora muito para carregar ou está sendo executado em outro computador. A execução do padrão

Proxy é criar um objeto que sua função é marcar o lugar gerenciando o acesso à o objeto alvo (GAMMA et al., 2000).

Conforme HORSTMANN (2007), o padrão *Proxy* é como se fosse o representante, um objeto que está substituindo outro objeto, também se explica que o objeto pode ser repositório de outro objeto.

2.1.3. Padrões Comportamentais

2.1.3.1. Interpreter

O padrão *Interpreter* é parecido com o padrão *Composite*, onde revela uma interface para grupos de itens e itens individuais todos eles comuns entre si. Para cada classe que compõe uma instância de *Interpreter* precisa formar uma regra de como as composições vão acontecer sendo que seu objetivo é coordenar as execuções, ou avaliações dessas coleções de regras (METSKER, 2004).

Para GAMMA et al. (2000), o padrão *Interpreter* representa como vamos estabelecer uma gramática para linguagens simples, executar princípios na linguagem e interpretar esses princípios.

2.1.3.2. Template Method

Conforme afirma (METSKER (2004), a intenção do *Template Method* é executar um algoritmo em um método, transferindo a definição de alguns procedimentos do algoritmo, para que outras classes possam reinterpretá-los.

O padrão *Template Method* também chamado de Gabarito de Método é representar o esqueleto de um algoritmo em uma operação qualquer, transferindo algumas etapas para as subclasses, propondo que as subclasses modifiquem certos passos de um algoritmo sem alterar a sua própria estrutura (GAMMA et al., 2000).

Para GUERRA (2013), o padrão *Template Method* se aplica na hora que se tem necessidade de descrever um algoritmo geral, que seria a sequência de objetivos para

cumprir uma exigência da aplicação sendo assim propondo que a estrutura que foi implementada ofereça um jeito para serem substituídos de maneira fácil.

2.1.3.3. Chain of Reponsibility

O padrão *Chain of Responsibility* é evitar o acoplamento do remetente de uma requisição ao seu receptor, para assim poder dar chance a mais de um objeto que consiga atender as solicitações (METSKER, 2004).

Segundo GAMMA et al. (2000), a intenção do padrão *Chain of Responsibility* é evadir o acoplamento do remetente da sua requisição ao seu próprio receptor, dando oportunidade a que mais de um objeto possa lidar com a solicitação, deixando os objetos receptores de forma encadeada e passando para frente a solicitação percorrendo a cadeia até que certo objeto lide com a requisição.

Para GUERRA (2013), *Chain of Responsibility* é um padrão com o propósito de criar uma cadeia de responsabilidades onde cada membro ou componente realiza uma operação de processamento de informação e passando a responsabilidade para o próximo da cadeia.

2.1.3.4. Command

O padrão *Command* estabelece assinaturas de métodos, podendo ser *execute()* ou *perform()*, nos deixando descrever várias implementações de interfaces e também nos permitindo encapsular solicitações como um objeto, para um modo em que podemos parametrizar clientes com solicitações diferentes, temporizar, registrar ou enfileirar (METSKER, 2004).

Para GUERRA (2013), o padrão *Command* fundamenta-se em representar operações como uma classe. O comando que é representado pela abstração, na forma de superclasse ou interface, interpreta um método que deve ser usado para realizar a operação.

2.1.3.5. Iterator

Conforme afirma METSKER (2004), às vezes necessitamos adicionar segurança de tipos e iteração ou se a demanda é criar um tipo novo de coleção, teremos que criar nosso próprio código *Iterador*, para percorrer a coleção. O padrão *Iterator* tem a intenção de oferecer um jeito de acessar os elementos dessa coleção de forma sequencial.

Propor uma maneira de acessar de forma sequencial elementos de um objeto cheio de dados sem mostrar a sua representação implícita (GAMMA el al., 2000).

2.1.3.6. Mediator

O padrão *Mediator* interpreta um objeto que encapsula a maneira de como um conjunto de objetos se relacionam. Isso propõe um baixo acoplamento, evitando que objetos se mencionam um ao outro no formato explícito, e nos permitindo mudar a interação deles de forma independente (METSKER, 2004).

Como afirma GUERRA (2013), o padrão *Mediator* foi proposto para criar uma classe que auxilia como mediadora entre os outros objetos. Em vez dos objetos enviarem pedidos e receber pedidos de vários outros, eles vão interagir apenas com a classe que está de mediador. Essa classe cuida dos pedidos recebidos e envia o pedido ou requisição para o objeto que deve receber.

2.1.3.7. Memento

Em certas situações temos o desejo de criar um objeto que foi removido anteriormente ou um usuário desfez operações e precisamos reverter para a versão ou ação anterior. O padrão *Memento* tem a intenção de propor o armazenamento e restauração do estado de um objeto (METSKER, 2004).

Para GAMMA et al. (2000), o padrão *Memento* possibilita armazenar o estado de um objeto para que posteriormente possa ser restaurado sem violar o encapsulamento desse objeto.

2.1.3.8. Observer

Uma solução é fazer com que os clientes sejam informados de quando o objeto sofre alguma mudança e deixar que eles mesmos realizem perguntas sobre o novo estado dele. A intenção do padrão *Observer* é definir uma dependência de um para-muitos, de modo que, quando um objeto mudar de estado, todos os seus dependentes sejam notificados e atualizados automaticamente (FREEMAN et al., 2009).

O objetivo deste padrão de projeto é definir uma relação de dependência de um para muitos entre um conjunto de objetos. Desta forma, quando um objeto muda de estado todos seus dependentes são notificados da tal mudança (METSKER, 2004).

2.1.3.9. State

Conforme MESTKER (2004), explica que a intensão do padrão *State* é repartir a lógica de estado através de classes que desempenham o estado de um determinado objeto. Isso faz com que seja permitido evitar *Ifs* complexos, deixando que o polimorfismo se responsabilize pela implantação da operação.

O padrão *State* é capaz de permitir que um objeto mude seu comportamento quando seu estado que esta interno mude também. Quando isso acontecer vai parecer que a classe desse objeto foi alterada (GAMMA et al., 2000).

2.1.3.10. Strategy

Uma estratégia é um plano, ou uma abordagem, para atingir um objetivo, dadas certas condições de entrada. Segundo FREEMAN et al. (2009), a intenção do *Strategy* é encapsular estratégias alternativas, ou abordagens, em classes separadas, cada uma das quais executa uma operação comum. A operação estratégica define as entradas e as saídas de uma estratégia, mas deixa a implementação para as classes individuais. As classes que realiza as várias abordagens implementam as mesmas operações e são, assim, intercambiáveis, apresentando a mesma interface aos clientes.

Para METSKER (2004), estratégia é uma missão, ou tipo de plano para atingir

um objeto, informada certas manipulações de entrada. A Intenção do padrão *Strategy* é encapsular métodos ou abordagens estratégicas e colocando tudo em classes separadas sendo que cada uma dessas classes executam suas operações normalmente. Essa operação estratégica interpreta as saídas e as entradas, deixando a parte da implementação para as classes separadas.

2.1.3.11. Visitor

Os desenvolvedores que fiscalizam o código da hierarquia não puderem ou não alterarem rapidamente o suficiente para atender suas necessidades poder ser que seja impossível estender o comportamento da hierarquia sem alterar suas classes. O padrão *Visitor* concede o encaixe de apoios as perspectivas de extensão sendo que outros desenvolvedores possam desejar estender este comportamento hierárquico. A intensão é propor uma nova operação de hierarquia sem alterar as classes (METSKER, 2004).

Para GAMMA et al. (2000), a intensão do padrão *Visitor* é representar ação a ser executada nos fundamentos de uma estrutura de objetos, define uma operação nova sem alterar as classes dos elementos sobre os quais ele atua.

Os padrões de projetos são observações que acontecem na prática, ajudam programadores do mundo inteiro a se comunicar da mesma forma, são soluções concretas definidas em pequenos nomes, propondo um *software* com menos erros, menos manutenção e tornando-o escalável. Para ser um padrão é necessário um problema, uma solução dentro de um contexto garantindo o máximo de desacoplamento entre classes, objetos e camadas.

3. ARQUITETURA DE SOFTWARE

Todo o *software* precisa ser projetado da melhor forma possível, para isso o projetista deve ter uma visão global do que esse sistema está composto, e se alterar uma parte do projeto qual impacto positivo ou negativo a solução final terá.

Segundo SILVEIRA et al. (2012), a arquitetura de *software* é uma maneira de ver o que foi implementado no projeto de *software*, verificando e entendendo qual o impacto que pode ocorrer se for realizada uma alteração. Análises desse tipo faz com que o arquiteto tenha uma visão global do sistema e as escolhas de como implementar influenciam o projeto por completo.

Conforme explica LARMAN (2005), a arquitetura de *software* é um aglomerado de decisões sobre como organizar um sistema de *software*, selecionar elementos da estrutura, da interface que o sistema constitui e o conjunto de como essa aplicação irá se comportar.

3.1. ARQUITETURA EM CAMADAS

As camadas de *software* têm o princípio de separar em partes as camadas físicas do sistema que segundo SILVEIRA et al. (2012), hoje em dia com a virtualização e a computação na nuvem é mais complicado explicar exatamente as barreiras físicas que podem existir em um sistema, mas os *tiers* também chamado de camadas, se caracterizam como componentes do sistema que poderiam ser executados em computadores separados. Servidores Web são considerados como *tiers*, assim como um servidor de *cache*, uma base de dados, servidor de aplicação, cliente remoto e outros exemplos.

Conforme explica FOWLER (2006), quando pensamos em um sistema com camadas imaginamos que o subsistema seja como a camada de um bolo, onde que cada camada realiza funções sem conhecer qual está abaixo dela, a camada superior aproveita serviços variados definidos pela camada abaixo sendo que a camada mais baixa não sabe da existência da camada mais alta.

Propor uma organização lógica em escala ampla de camadas discretas com diferentes responsabilidades e relacionadas, mantendo os interesses separados e coerentes fazendo com que as camadas abaixo sejam apenas de serviços ou de baixo

nível e as camadas superiores seja específicas da aplicação, apenas existindo acoplamento e colaboração entre as camadas superiores (LARMAN, 2005).

3.1.1. Arquitetura Monolítica

Arquitetura baseada em um *tier* tem sido popular no tempo dos mainframes onde o objetivo era centralizar todos os processos no mesmo lugar sendo que eles eram ao mesmo tempo interface e persistência do aplicativo, a sua vantagem na época foi a diminuição da latência e melhor desempenho sendo que não era necessário realizar requisições através da rede para procurar informações, mas por outro lado a sua desvantagem é a de não ser escalável, pois era incapaz de compartilhar a aplicação com outros usuários de forma arbitrária e arquiteturas centralizadas podem ter problemas de disponibilidade (SILVEIRA et al, 2013).

Segundo BOND et al. (2003), a maioria dos sistemas monolíticos eram aplicativos contendo todas as suas funcionalidades em apenas um único local, todo o código era encontrado em um único arquivo chamado também de código espaguete, tornando difícil a manutenção e a evolução do sistema. Na figura 02 exemplifica essa questão da arquitetura de *software* monolítica:

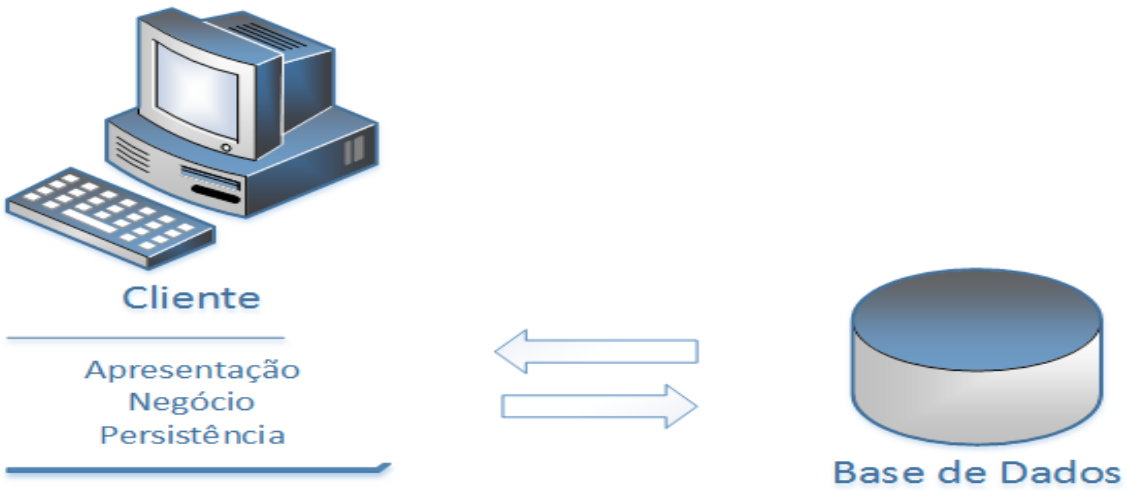

Figura 02 – Arquitetura com uma camada
Fonte: Baseado em BOND et al. (2003)

3.1.2. Arquitetura em Duas Camadas

As arquiteturas em dois *tiers,* tem como exemplo aplicações *Desktop* conectadas a uma base de dados centralizada, que segundo SILVEIRA et al. (2013), na camada do cliente está situado todo o código pertencente ao sistema, sendo assim chamados de *fat client*. A desvantagem é que pode surgir uma carga excessiva de processos na parte do cliente, dificuldades em manter tudo integrado e necessidade de segurança. Para diminuir o problema de processamentos elevados na parte do cliente em alguns casos mudamos as regras de negócio para a parte do banco de dados isso se chama *stored procedures*.

Conforme explica FOWLER (2006), a noção de camadas nas aplicações surgiu no ano de 1990, com o princípio chamado cliente-servidor onde a parte do cliente mantinha todos os códigos da regra de negócio e a parte do servidor apenas a base de dados relacional. Com a necessidade de cálculos e validações, os códigos eram escritos na parte do cliente, mas isso acabava dificultando a manutenção quando o sistema se tornava mais complexo, e uma das alternativas era escrever as regras de negócio na base de dados. Na figura 03 exemplifica essa questão da arquitetura de *software* em duas camadas:

Figura 03 – Arquitetura com duas camadas
Fonte: Baseado em SILVEIRA et al. (2012)

3.1.3. Arquitetura em Três Camadas

Como afirma SILVEIRA et al. (2012), toda a parte de regras de negócio seria executada na camada intermediária entre a camada do cliente e a camada de armazenamento de dados, o cliente realiza requisições para o servidor que processa a regra de negócio e depois ele delega essa requisição de manipular os dados para a base de dados. Por isso não faz mais sentido todo o processamento das regras de negócio na camada do banco de dados. Mas ainda sim permanece o problema de disponibilidade e de escalabilidade.

A arquitetura de *software* em três camadas teve mais força na chegada da Web, onde passaram a instalar *softwares* cliente-servidor usando o navegador devido ao fato de que criar páginas Web eram menos atadas à linguagens SQL sendo mais adaptável para a terceira camada (FOWLER, 2006). Na figura 04 exemplifica essa questão da arquitetura de *software* em três camadas:

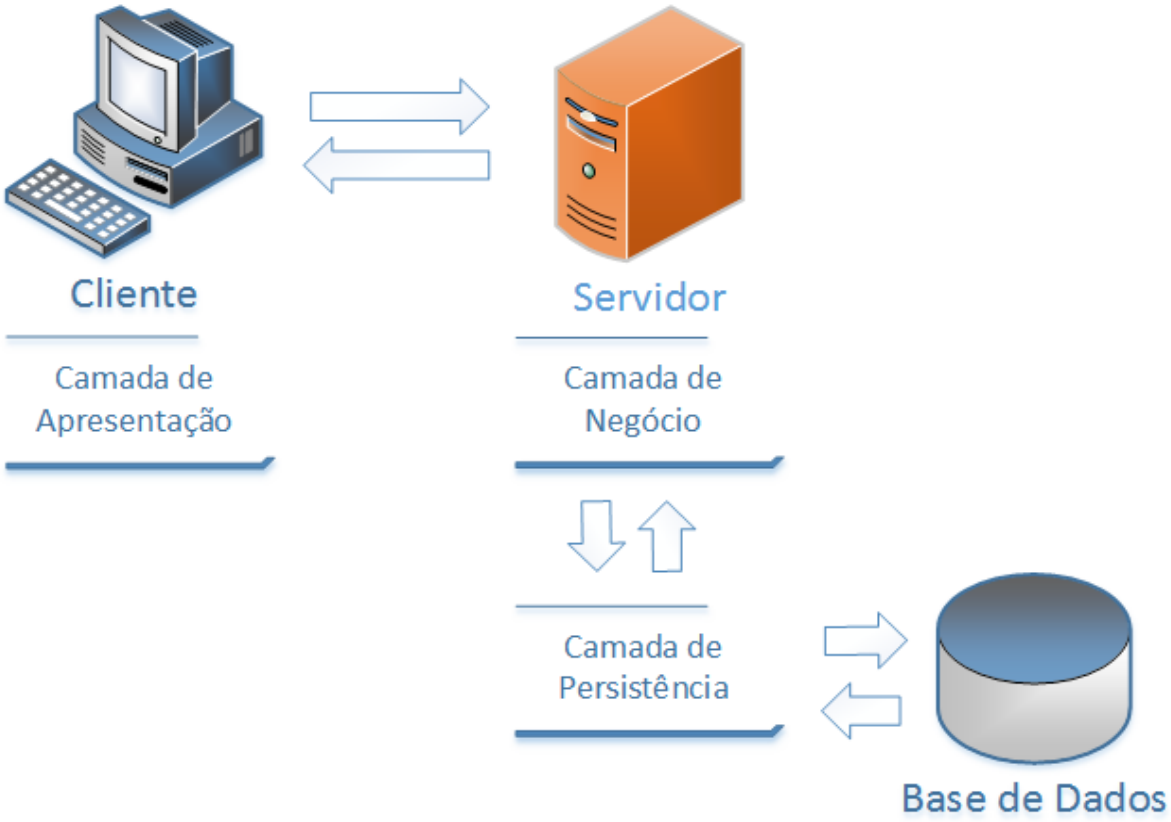

Figura 04 – Arquitetura com três camadas
Fonte: Baseado em SILVEIRA et al. (2012)

3.1.4. Arquitetura em N. Camadas

A arquitetura com várias camadas é uma variação da arquitetura em três camadas, SILVEIRA et al. (2012), explica que a as funcionalidades da arquitetura em N. camadas também chamada de *N-tiers*, é resolver problemas de disponibilidade e de escalabilidade. Existem diversos exemplos de adicionar novos *tiers*, como servidores de aplicações Web, servidores de *caches* e de aplicação. Os pontos negativos dessa arquitetura é que quando adicionar mais camadas mais complexo se torna o gerenciamento e a manutenção de toda a estrutura, é apenas recomendável quando há requisitos muito fortes de disponibilidade e escalabilidade.

A arquitetura em N. camadas pode ser dividida fisicamente ou logicamente, podendo separar fisicamente usando servidores para cada camada, isso aumenta a complexidade e o custo da estrutura e do desenvolvimento do *software,* podemos separar logicamente em apenas um servidor contendo todas as camadas separadas de forma lógica (BOND et al., 2003). Na figura 05 exemplifica essa questão da arquitetura de *software* em N. camadas:

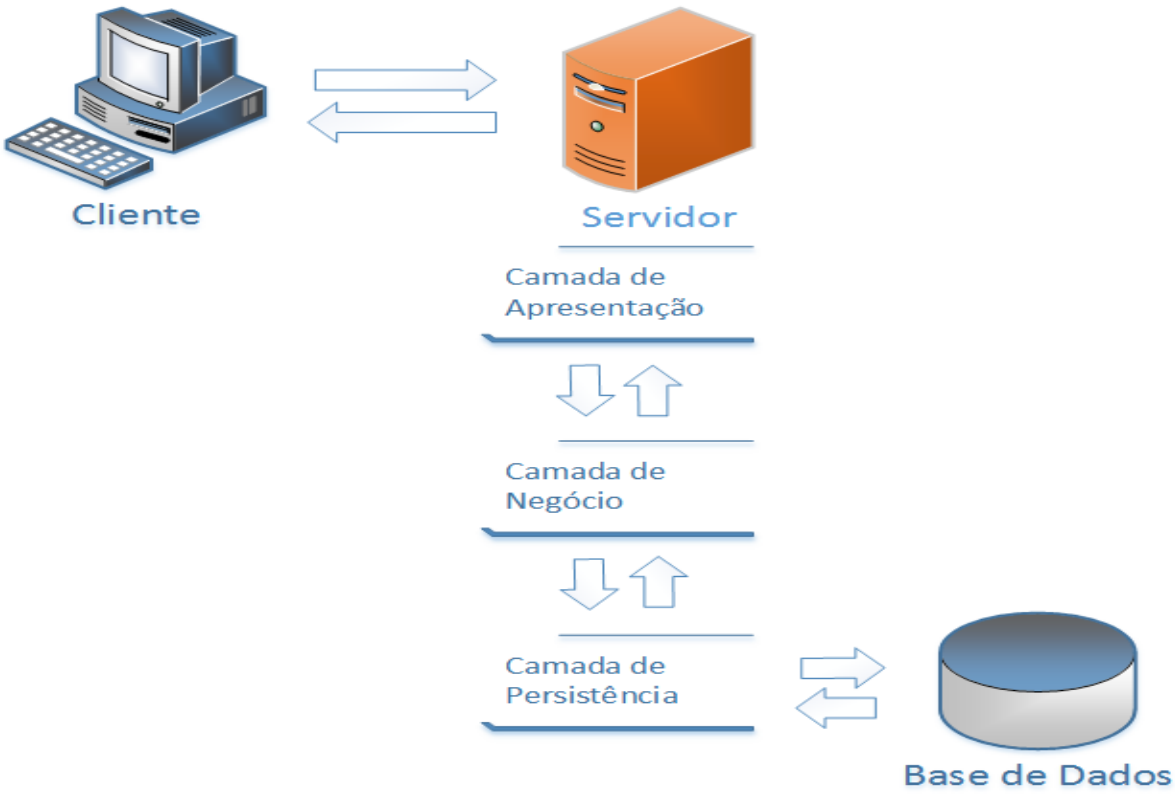

Figura 05 – Arquitetura com N camadas
Fonte: Baseado em BOND et al. (2003)

Segundo BOND et al. (2003) as camadas de apresentação, negócio e persistência são camadas lógicas e que cada uma delas tem sua responsabilidade sendo que a camada de apresentação é o que interage com o usuário do sistema e como os dados são mostrados para ele, a camada de negócio é a parte que cuida do negócio, são regras ou funções específicas do sistema e a camada de persistência é responsável pela conexão com a base de dados persistindo informações ou alimentando o sistema com as informações persistidas.

O projetista de sistemas tem uma necessidade constante em desenvolver aplicativos maduros, para isso é necessário que a aplicação seja projetada da melhor forma possível, com a ajuda da arquitetura de *software* isso é possível pois proporciona uma visão global de toda a parte do sistema, e ao realizar alterações qual é o impacto que isso pode trazer a solução final de *software*.

4. FRAMEWORK DE DESENVOLVIMENTO

Os *frameworks* aprimoram a modularidade e o encapsulamento dos detalhes da implementação através de interfaces estáveis, ajudando a criar *softwares* com mais qualidade reduzindo o esforço necessário, facilitando o entendimento e mantendo a sua existência (FAYAD, 1997).

Conforme destaca SCHMITZ (2009), a definição para *frameworks* na literatura de vários autores é denominada como muito abstrata, mas propõe recurso essencial básico para desenvolver sistemas através de um conjunto de arquivos que próximos um dos outros automatizam trabalhos de caráter repetitivo.

Para LISBOA (2009), ao utilizar um *framework* de desenvolvimento, estamos reaproveitando o pensamento de outra pessoa e que esse reaproveitamento cita a orientação a objetos sendo que reaproveitar o código não é da programação orientada a objetos e sim, que seus paradigmas podem fazer com que se a implemente.

Os pensamentos de HORSTMANN (2007), explica que *frameworks* são classes que executam de forma essencial e cooperativa o domínio de problemas descritos, elas são classes que incrementam funções para um determinado entendimento, são padrões múltiplos.

Os *frameworks* são como *softwares* que não estão completos, segundo GUERRA (2013) é necessário o preenchimento com partes mais específicas de um sistema para poder funcionar, sendo assim diferente dos padrões de projetos, pois é de fato um sistema e não uma ideia como é um padrão.

São como um esqueleto, um conjunto de muitos padrões de projetos e soluções implementadas parcialmente pelos criadores do *framework* para o programador utilizar como desejar. O *framework* Ruby on Rails está a mais tempo no mercado, o Play framework é mais atual e segue a mesma filosofia do *framework* da linguagem Ruby.

4.1. RUBY ON RAILS

O Ruby on Rails é um *framework* escrito na linguagem ruby como explica FUENTES (2013), utiliza a estrutura chamada MVC – *Model View Controller* também chamado de Modelo, Apresentação e Controle, bastante apropriado para projetar sistemas Web, sendo que o Rails assim também chamado, é *opensource* criado por David Heinemeier Hanson no ano de 2003.

Segundo TATE (2006), talvez o Ruby on Rails seja um dos projetos mais importantes dos últimos 10 anos, é um dos *frameworks* mais produtivos para se desenvolver sistemas Web, sendo baseado na linguagem Ruby permitindo que você crie *software* na estrutura modelo-visualização-controlador.

Uma das técnicas mais exploradas na engenharia de *software* é a criação de componentes reutilizáveis, isso diminui o tempo de desenvolvimento e erros na hora de codificar.

O Ruby on Rails é composto por 5 módulos independentes:

- *Active Record* é o responsável por integrar os objetos com as tabelas do banco de dados;
- *Action Controller* se responsabiliza por coordenar a interação entre o usuário, as visões e o modelo. Também responsável pelas rotas para ações internas das *controllers*;
- *Action Mailer* é responsável pelos serviços de entregar e receber e-mails;
- *Active Support* é uma coleção de classes utilitárias e extensões de bibliotecas muito útil para o *framework* Ruby on Rails;
- *Action View* é responsável por conter toda as funcionalidades necessárias para renderizar *views*.

Por ser projetado em arquitetura modelo-visualização-controlador, Rubyonrials (2007) afirma que as funções do sistema desenvolvido em Ruby on Rails se inicia por um pedido no navegador ou também chamado de *request*, isto é uma função que o

cliente solicita para o sistema, o *software* processa esse pedido e retorna uma resposta para o solicitante. A Figura 06 explica a visão geral do ciclo de vida das requisições no *framework* Ruby on Rails:

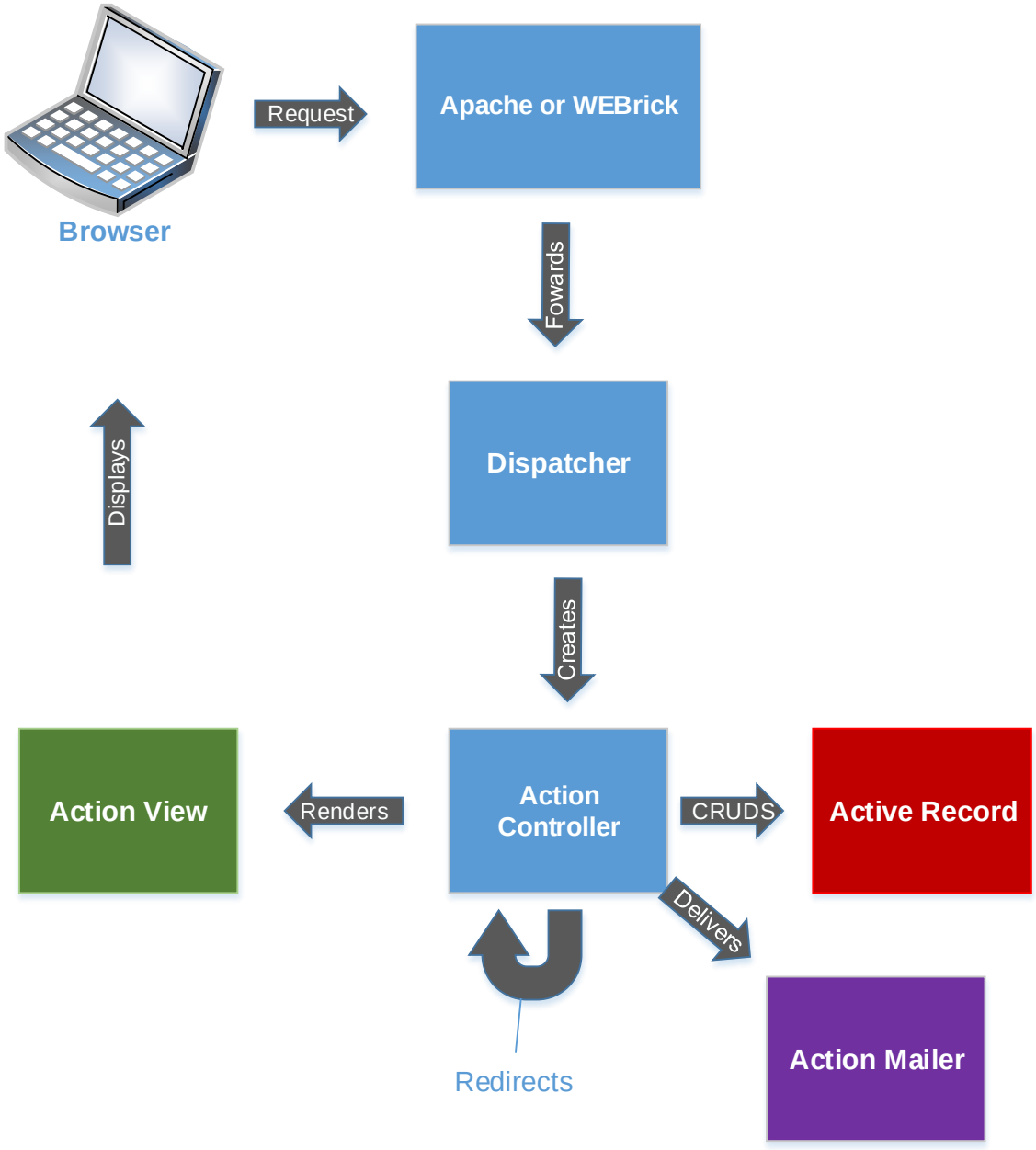

Figura 06 – Ciclo de Requisições Ruby on Rails
Fonte: Baseado em Rubyonrails (2007)

Ao realizar o pedido de uma página, o navegador faz uma chamada ao servidor Apache, sendo encaminhado para o *dispatcher.* Segundo Rubyonrails (2007), o *dis-*

patcher é responsável por transformar a URL do navegador em uma URL que o *framework* Ruby on Rails possa entender. A *Action Controller* é invocada para decidir o que fazer com o pedido recebido, se necessário alguma interação com o banco de dados o Active Record se responsabiliza pelas ações caso contrário se o *Action Mailer* é invocado se o pedido seja o de enviar e-mails, e a resposta da requisição é processada pelo *Action View.*

4.2. PLAY FRAMEWORK

Segundo Playframework (2014), no ano de 2007 os projetistas estavam pensando em criar uma jeito de desenvolver aplicações web de maneira menos complicada, o Play framework era um projeto interno da Zenexity situado da cidade de Paris, foi influenciado pela maneira de fazer projetos Web sempre focando na produtividade do desenvolvedor, no ano de 2009 foi decidido compartilhar essas ideias com a comunidade como projeto de código aberto, o projeto ganhou força depois do *feedback* positivo na comunidade e após 2 anos o Play framework teve duas versões e uma comunidade ativa de 4000 pessoas com constante crescimento e muitas aplicações rodando em todo o globo.

O Play framework reinterpreta o desenvolvimento Web no Java, BOAGLIO (2014), afirma que o foco do Play framework é na produtividade do desenvolvedor em arquiteturas RESTful, interface HTTP simplificada, aproveitando todo o poder da máquina virtual do Java.

Como explica SOUZA (2014), o Play framework não segue os padrões de especificações do JAVAEE isso o fez levar para um caminho onde se oferece várias tecnologias para diferentes partes da sua aplicação tudo de forma integrada, conhecido também como *framework Full Stack* tornando o assim um dos mais produtivos do mercado.

O Play framework não segue os padrões do JAVAEE, ele integra várias soluções disponíveis no mercado para projetar *softwares* com menos complicação e agilidade.

Assim como o Ruby on Rails o Play Framework também é projetado na camada modelo-visualização-controlador, na Figura 07 explica a visão geral de como funciona a parte de solicitações na camada MVC no *framework* Play framework:

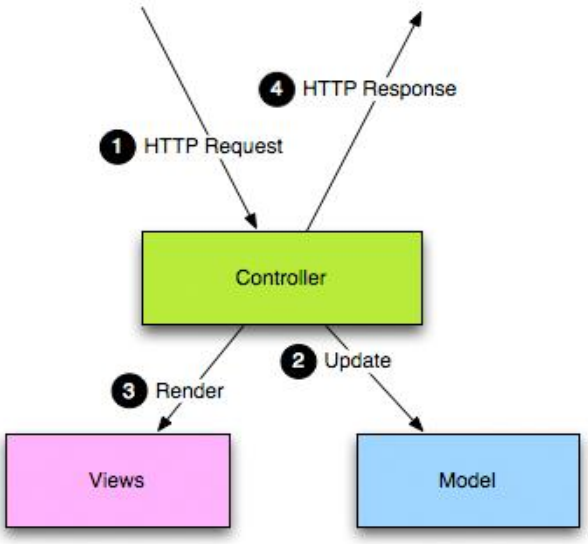

Figura 07 – Estrutura MVC Play Framework
Fonte: Baseado em Playframework (2014)

O Play framework segue o padrão MVC aplicada à arquitetura Web, Playframework (2014), afirma que o controlador é uma classe Java onde os métodos são ações públicas e estáticas, é o ponto de entrada quando uma solicitação HTTP é recebida. O método de ação é extrair os dados relevante da solicitação, lê ou atualiza os objetos do modelo e depois envia de volta um resultado que está envolvida em uma resposta HTTP. A camada de modelo é um conjunto de classes Java usando todos os recursos orientados a objetos disponíveis a partir da linguagem Java. Contem estrutura e operações para que a aplicação consiga operar com dados. A parte da visão, são geradas usando um sistema de *templates* fornecida pelo Play. O controlador recebe alguns dados a partir da camada de modelo e em seguida aplica esse modelo assim decorando seus objetos.

A controller responde a eventos ou ações do usuário como alterações nos dados ou compilar uma página, a parte da *view* é responsável pela interação entre o utilizador e o sistema que também pode chamado de interface e a parte da *Model* se

responsabiliza pelo domínio da informação, os dados brutos. A Figura 08 explica a visão geral do ciclo de vida das requisições no *framework* Play Framework:

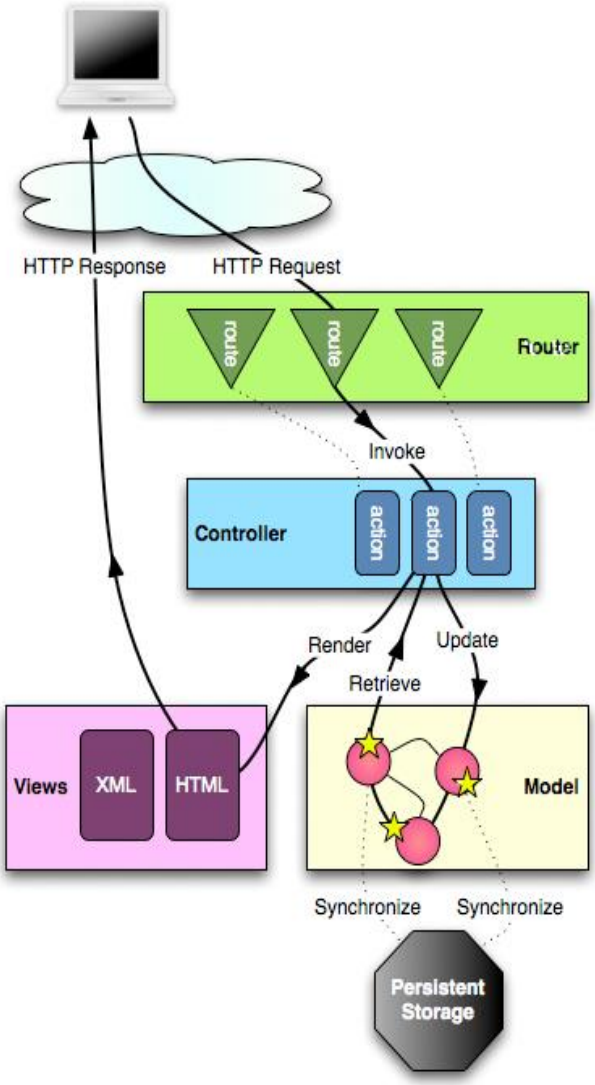

Figura 08 – Ciclo de Requisições Play Framework
Fonte: Baseado em Playframework (2014)

O Play Framework é totalmente estático e orientado a pedido/respostas, todas as solicitações seguem o mesmo caminho, Playframework (2014) explica que quando o pedido é recebido pelo *framework* o componente de rotas realiza uma busca para a rota mais específica compatível com o pedido HTTP recebido, quando a rota for encontrada, o método de ação que se localiza na *controller* é invocado e a aplicação

executa o código retornando uma *view* complexa em forma de HTTP, e se for uma ação de alterar dados brutos, um arquivo de modelo é renderizado e o resultado da ação é então retornada em forma de HTTP novamente.

Na figura 08 demonstra o ciclo de vida do Play Framework que quando o navegador realiza um pedido de requisição, essa requisição vai para o arquivo *router,* o Play framework realiza uma busca e verifica se a requisição se encaixa corretamente com a encontrada no arquivo de rotas, ao encontrar a requisição solicitada é invocada uma ação para a *controller* respectiva da requisição, se essa ação é de renderizar uma página o framework compila a página HTML e retorna para o navegador, caso a ação seja de atualizar, buscar informações e salvar a Model do Play framework se responsabiliza em se comunicar com a base de dados.

5. METODOLOGIA DE COMPARAÇÃO DE FRAMEWORKS

Na literatura, são dadas poucas orientações para a comparação de *frameworks*; algumas comparações são feitas com base no projeto arquitetônico do próprio *framework* (PRINJIN, 2006).

Segundo FAYAD (1997) uma das listas de características de comparar *framework* é dada para *frameworks* de aplicação empresarial, essas características incluem modularidade, escalabilidade, independência de plataforma e capacidade de extensão.

Observa-se duas vertentes na análise comparativa entre os *frameworks* Play framework e Ruby on Rails que na visão de FAYAD (1997), os *frameworks* são comparados através da sua escalabilidade, independência de plataforma e sua capacidade de extensão, na visão de PRINJIN (2006), algumas comparações são feitas através de como foi projetado arquitetonicamente cada *framework*.

Neste trabalho decide-se a melhor vertente de comparar *frameworks* é a visão de FAYAD (1997), onde são comparadas várias características em comum entre as duas ferramentas para que assim demonstre de maneira clara a capacidade que cada *framework* possui.

5.1 ESCALABILIDADE

A escalabilidade é a habilidade que o sistema tem de manter o seu funcionamento ou respostas dos objetivos mesmo quando surgir uma nova demanda de adicionar novas funções a ele (SMITH, 2001).

Para falar sobre escalabilidade devemos primeiramente definir em qual contexto é cabível que segundo FAYAD et al. (2001), no contexto da engenharia de *software* é de propriedade reduzir ou aumentar o âmbito dos métodos, processos, e gestão de acordo com o tamanho do problema.

5.2. INDEPENDÊNCIA DE PLATAFORMA

O termo Independência de plataforma está se referindo ao *software* que pode ser executado em vários equipamentos diferentes, é um conjunto de tecnologias repleta de funções coerentes, que podem ser utilizadas, segundo MELLOR (2004), a plataforma se define em descrever um ambiente de execução para determinado conjunto de modelos.

5.3. CAPACIDADE DE EXTENSÃO

Um *framework* aumenta sua capacidade de extensão fornecendo métodos ganchos de maneira explícita, assim permitindo que as aplicações possam expandir suas interfaces estáveis. Os métodos ganchos desacoplam sistematicamente os comportamentos e interfaces estáveis sendo necessário para a instanciação de uma aplicação num contexto em particular, essa capacidade de extensão é essencial para garantir uma customização rápida de novos recursos, e serviços para o aplicativo (FAYAD, 2001).

5.4. MODULARIDADE

A modularidade ajuda a melhorar a qualidade do *software*, localizando impactos e implementando alterações, o que reduz o esforço necessário mantendo o entendimento da aplicação já existente (FAYAD, 1997).

Modularidade é o conceito mais comum de separar interesses, segundo PRESSMAN (2011), o sistema é dividido em partes também chamados de módulos, que quando integrados satisfazem os requisitos de um problema tornando o *software* gerenciável e mais fácil de se desenvolver.

Pouco embasamento teórico é encontrado sobre as metodologias de comparação de *frameworks,* com os estudos de FAYAD (1997), existem 3 tipos de *frameworks* e um deles é chamado de *frameworks* para aplicações empresariais onde que para compará-los existem 4 características, sendo eles a escalabilidade, independência de plataforma, capacidade de extensão e modularidade. A escalabilidade é a habilidade que o sistema tem de adicionar novas funções sem prejudicar as já existentes ou implementadas, devemos sempre pensar em projetar sistemas que futuramente novas funções serão adicionadas devido a demanda de novas funcionalidades requeridas pelos clientes ou o mercado. Independência de plataforma refere-se na questão que o sistema funcione em vários equipamentos diferentes sem precisar modificar bruscamente as funções já existentes.

A Capacidade de extensão segundo GUERRA (2013), é uma das caraterísticas mais utilizadas pelos *frameworks,* onde essa capacidade é adquirida pelos métodos ganchos, que promove a especialização de comportamento ajudando o sistema a ter novas lógicas de execução. A modularidade é uma característica bem comum de separar interesses, onde é dividir o sistema em partes que quando juntos satisfazem os requisitos de um problema tornando o mesmo gerenciável e fácil de projetar.

6. COMPARATIVO ENTRE O RUBY ON RAILS E PLAY FRAMEWORK

Os *frameworks* Play Framework e Ruby on Rails possuem vários atributos em comum, como a arquitetura em modelo-visualização-controlador, suporte à validação HTML, internacionalização, suporte a vários sistemas de banco de dados, suporte a parte de testes de sistema a vários sistemas operacionais. Neste capítulo serão comparados alguma das suas principais características seguindo o propósito de FAYAD (1997) onde é a Capacidade de Extensão, Escalabilidade, Modularidade e a Independência de Plataforma. Essas características serão apresentadas em forma de tabela:

Tabela 01 – Estrutura comparativa entre Play Framework e Ruby on Rails
Fonte: (do autor)

Características	Play Framework	Ruby on Rails
Capacidade de Extensão		
Internacionalização	I18n	I18n
Escalabilidade		
Suporte a banco de dados	Mysql, PostgresSQL, MongoDB(no-sql), ScalikeJDBC	Mysql, PosrtgresSQL, SQLite
Modularidade		
Suporte a testes de *software*	Unitário, TDD	Unitário, TDD
Validação das páginas HTML	Páginas de cadastros	Página de cadastros
Independência de Plataforma		
Suporte aos sistemas operacionais	Windows, Linux, Mac OS	Windows, Linux, Mac OS
Linguagens de Programação	Java e Scala	Ruby
Suporte a Deploy na Nuvem	Heroku, Google App Engine	Heroku, Phusion Passenger, JRuby, Capistrano
IDEs de Desenvolvimento	IntelliJ Idea, Eclipse, Sublime Text, Tem sua própria IDE Integrada através da type safe	IntelliJ Idea Rubymine, Aptana Studio 3, Ruby on Steel, Netbeans, Editores de Texto

Após a verificação da tabela 01, analisamos uma estrutura comparativa proposta pelo autor seguindo os princípios do FAYAD (1997) levando em conta características comuns e diferentes entre os dois *frameworks* separados por módulos chamados de capacidade de extensão, escalabilidade, modularidade e independência de plataforma. Abaixo descreve-se as caraterísticas que foram comparadas e os critérios correspondentes:

6.1. INTERNACIONALIZAÇÃO

Também conhecido como I18n, é o processo de adaptar um *software* para um idioma definido, muitos métodos utilizados em sistemas como a maioria dos padrões de projetos retornam mensagens e a maior parte dessas mensagens são por padrão no idioma inglês. O propósito desse conceito é que o usuário não seja forçado a aprender o idioma inglês para utilizar o sistema.

A Internacionalização são muito comuns em sistemas WEB, pois pessoas de vários países podem utilizar o sistema sem precisar implementar uma nova página para cada idioma existente, esse é um meio de adaptar a sua aplicação para diferentes linguagens e permitir as diferenças regionais isso é um ponto a ser avaliado se seu projeto tem tendências a ter mais de um idioma.

6.2. SUPORTE A VÁRIOS BANCO DE DADOS

Existem várias bases de dados no mercado, sendo eles de código aberto, proprietário, NoSQL e objeto relacional, ter suporte à os bancos de dados mais utilizado no mercado como por exemplo MySQL, PostgresSQL e o MongoDB é importante e pode garantir que o *framework* possa ser usado em várias situações e também ser um diferencial na hora da escolha da ferramenta ou de uma demanda na hora de projetar um sistema.

6.3. SUPORTE A TESTES DE SOFTWARE

Conforme novas funções vão sendo implementadas no sistema, a chance de funções já criadas não funcionarem da maneira que funcionava é muito elevada. Tanto o Ruby on Rails quando Play Framework é possível realizar testes unitários e testes TDD. O teste unitário por sua vez tem a função de avaliar como está a entrada e saída de dados de um sistema ou um método por exemplo, consiste em avaliar se os dados a serem retornados estão coerentes do esperado, teste unitário é a menor parte que pode ser testada em um sistema como uma função dentro de uma *controller*. Os testes TDD são desenvolvimento orientado a testes, serão realizados testes para cada funcionalidade que vai existir no sistema, gerando um clico de repetições para quando ser criado as funcionalidades verdadeiras possam passar pelos testes que foram criados anteriormente.

6.4. SUPORTE A VÁRIOS SISTEMAS OPERACIONAIS

Os sistemas operacionais mais utilizados no mercado são Microsoft Windows, Linux e Apple Mac OS, essa característica é importante pois não é necessário que o programador mude de sistema operacional apenas para o uso do determinado *framework*.

6.5. VALIDAÇÃO DAS PÁGINAS HTML

Ter suporte à validação de páginas HTML em poucas linhas de código é uma solução indispensável para a agilidade na hora de desenvolver um sistema. Se tratando de uma aplicação que vai gerar dados através de uma interface com o usuário, é de extrema importância, para não ocorrer de serem salvos dados inválidos. Essas validações são decoradas em uma linha de código no próprio atributo de uma classe, sendo eles não nulos, apenas números, apenas caracteres ou a validação de e-mail.

6.6. LINGUAGENS DE PROGRAMAÇÃO

Para o *framework* se expandir rapidamente no mercado é necessário que o mesmo seja projetado em uma linguagem bastante conhecida no mundo do desenvolvimento de *software*, isso promove o conhecimento e faz com que os engenheiros de *software,* escolham a ferramenta por sua linguagem de programação ser utilizada por uma grande fatia de empresas na área de desenvolvimento de sistemas.

6.7. SUPORTE A DEPLOY NA NUVEM

Serviços de armazenamento na nuvem são um diferencial de mercado, pois através dessa solução é possível armazenar o código do *software* e fazer com que o mesmo seja executado em um servidor que esteja na nuvem, isso promove a agilidade em apresentar um protótipo do sistema já em pleno funcionamento ao cliente com apenas algumas configurações no *framework*. Um dos processos mais comuns atualmente é realizar alterações nas funções do sistema e após o término salvar as configurações e executar novamente a aplicação que está na nuvem com as alterações feitas, tudo isso com pouco tempo gasto.

6.8. IDEs de DESENVOLVIMENTO

Conhecido como Ambiente de desenvolvimento Integrado, a IDE funciona como um editor de texto, com funções especiais para ajudar os projetistas na hora de digitar o código fonte do *software,* diminuindo os erros de códigos digitados, melhorando o processo de *debug* e propondo várias outras soluções, como por exemplo *plugins* para se integrar melhor ao *framework*, sistema de versionamento de código sendo por um lado versões de comunidades e por outro lado versões pagas.

Na tabela 01 demonstra algumas das características mais comuns entre os dois *frameworks*, existem muitas outras características que não foram mostradas neste trabalho devido a necessidade de maior aprofundamento no tema em questão. Mas

as declaradas têm visão suficiente para ajudar o programador e escolher qual *framework* utilizar dependentemente da demanda nas funções que o sistema final terá. Na característica de internacionalização os dois *frameworks* tem suporte a mesma sendo ela a I18n, em suporte a vários banco de dados o Play Framework leva vantagem em relação ao Ruby on Rails por ter mais opções como por exemplo o banco de dados MongoDB pouco utilizado em comparação ao MySQL ou ao PostgreSQL por ter um conceito diferente por não ser objeto relacional mas com grande potencial e ser escalável, na característica de suporte a testes de *software* os dois *frameworks* atendem a necessidade de testes unitários e testes TDD, em suporte a vários sistemas operacionais as duas ferramentas atendem aos mais utilizados no mercado, sendo eles o Microsoft Windows, Linux e o Mac OS, validações de páginas HTML é indispensável caso o programador queira ganhar tempo e os dois *frameworks* atendem a essa necessidade, em linguagens de programação é possível programar no Play Framework em Java e Scala e no Ruby on Rails apenas na linguagem Ruby, suporte a *deploy* na nuvem é um grande diferencial e o Ruby on Rails se sai na frente por ter mais soluções no mercado e promover mais facilidade e opções nessa questão, para programar não é necessário uma IDE, apenas um editor de texto mas para facilitar e ter mais integridade com o código digitado as IDEs são uma boa ferramenta para garantir menos erros de código.

7. CONCLUSÃO

Neste trabalho foi utilizado a metodologia de comparação de *frameworks* com princípios em aplicações empresariais para ajudar o programador a decidir qual ferramenta utilizar caso escolha entre Ruby on Rails e Play Framework. As características que cada ferramenta apresenta são semelhantes seguindo o mesmo conceito de desenvolver sistemas WEB de forma rápida e com o menor número de complicações possíveis.

Os *frameworks* têm uma grande abrangência no quesito de ferramentas e soluções em que atendem, em suporte a variedades de banco de dados o Play framework é a melhor escolha caso o projetista de *software* tenha uma demanda de desenvolver um sistema em base de dados NoSQL, atualmente pouco utilizado no mercado, porém com conceitos diferenciados das bases de dados objeto relacional. Outros fatores importantes para a escolha do Play framework consistem na questão de se programar tendo duas opções em linguagens de programação, sendo elas em Java e Scala, ter suporte a um maior número de servidores de aplicação e tendo também seu próprio ambiente de desenvolvimento integrado.

O Ruby on Rails se destaca por estar mais tempo no mercado, isso o deixou mais desenvolvido em relação ao Play Framework, o Rails foi projetado na linguagem Ruby onde segue o conceito de não repetir códigos que oferecem o mesmo resultado, tendo mais variedades para realizar um *deploy* da aplicação na nuvem caso isso seja um requisito na hora de projetar o *software*.

O fato é que o Ruby on Rails é um dos *frameworks* de código aberto mais utilizado atualmente, com mais tempo no mercado, isso o tornou mais poderoso e estável comparado ao Play Framework, com pequenas buscas na internet o programador acaba encontrando uma grande quantidade de soluções prontas em Rails isso também agiliza na hora de desenvolver um sistema, sendo que em Play Framework já é mais difícil por estar a menos tempo no mercado e ser menos popular.

Com pouco embasamento teórico sobre o assunto de comparação de *frameworks* fica claro que as grandes empresas de desenvolvimento de *software* acabam optando pelo Ruby on Rails por ele suprir praticamente todas as necessidades de se projetar um sistema robusto e escalável.

8. TRABALHOS FUTUROS

A partir da leitura desse comparativo entre os *frameworks*, o próximo passo deve ser um aprofundamento em outros *frameworks* de desenvolvimento WEB que por ventura não foram abordados neste trabalho como o Spring MVC, Django, Jersey e outros, nessa pesquisa devem ser abordadas outras características de cada um para que assim a pesquisa possa ser aprofundada e aperfeiçoada.

Com o aprofundamento da pesquisa, podemos chegar ao conhecimento de outros tipos de desenvolvimento, como exemplo podemos citar o *mobile* que foi desenvolvido para aparelhos portáteis.

Neste trabalho desenvolvido, a comparação entre os *frameworks* foi realizada com muito sucesso, porém seria muito importante que trabalhos futuros se aprofundassem em cada *framework*, de maneira mais extensa, para que esse comparativo possa ser ainda mais completo e eficaz.

9. REFERÊNCIAS BIBLIOGRÁFICAS

BOAGLIO, Fernando. **Play Framework** – Java para web sem servlets e com diversão. São Paulo: Casa do Código, 2014.

BOND, Martin et al. **Aprenda J2EE em 21 Dias**. São Paulo: Pearson Education do Brasil, 2003.

FAYAD, Mohamed E.; SCHMIDT, Douglas C.. **Object-oriented application frameworks.** Communications of the ACM,1997.

FAYAD, Mohamed E.; LAITINEN, Mauri; WARD, Robert P.. **The Problem with Scability** – In software, very few projects remains of constant magnitude. Comunications of the ACM, 2001.

FOWLER, Martin. **Padrões de Arquitetura de aplicações corporativas**. Porto Alegre: Bookman, 2006.

FREEMAN, Eric et al. **Padrões de Projetos**. 2. Ed. Rio de Janeiro: Alta Books, 2009.

FUENTES, Vinícius Baggio. **Ruby on Rails** – Coloque sua Aplicação Web nos Trilhos. São Paulo: Casa do Código, 2013.

GAMMA, Erich et al. **Padrões de Projeto** – Soluções Reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000.

GUERRA, Eduardo. **Design Patterns com Java** – Projeto Orientado a Objetos guiado por Padrões. São Paulo: Casa do Código, 2013.

HORSTMANN, Cay. **Padrões de Projetos orientados a objetos.** 2. Ed. Porto Alegre: Bookman, 2007.

LARMAN, Craig. **Utilizando UML e Padrões** – Uma introdução à análise e ao projeto orientado a objetos e ao desenvolvimento iterativo. Porto Alegre: Bookman, 2005.

LISBOA, Flávio Gomes da Silva. **Zend Framework** – Componentes Poderosos para PHP. São Paulo: Novatec, 2009.

MELLOR, Stephen J. et al. MDA Distilled: Principles of Model-Driven Architecture. Addison Wesley, 2004.

METSKER, Steven John. **Padrões de Projetos em Java**. Porto Alegre: Bookman, 2004.

PLAYFRAMEWORK. **Play Framework Philosophy.** 2014. Disponível em: https://playframework.com/documentation/2.3.x/Philosophy. Acesso em: 27/04/2015.

PRESSMAN, Roger S. **Engenharia de Software** – Uma Abordagem Profissional. Porto Alegre: Bookman, 2011.

PRINJIN, Tim. **Framework software quality analysis**. Dissertação de Mestrado, Universidade do Amsterdam, 2006.

RUBYONRAILS. **Ruby on Rails Guide.** 2009. Disponível em: http://guides.rubyonrails.org/v2.3.11/getting_started.html. Acesso em: 10/06/2015.

SCHMITZ, Daniel Pace. **Dominando Flex e Zend.** São Paulo: Canal6, 2009.

SHALLOWAY, Alan; TROTT, James R.. **Explicando Padrões de Projeto** – Uma nova Perspectiva em Projeto Orientado a Objeto. Porto Alegre: Bookman, 2004.

SILVEIRA, Paulo et al. **Introdução à Arquitetura e Design de Software** – Uma visão sobre a plataforma java. Rio de Janeiro: Elsevier, 2012.

SMITH, Connie U.; WILLIAMS, Lloyd G. **Performance Solutions:** A practical Guide to Creating Responsive, Scalable Software. Boston: Addison-Wesley, 2001.

SOMMERVILLE, Ian. **Engenharia de Software**. 8. Ed. São Paulo: Addison Wesley, 2007.

SOUZA, Alberto. **Play Framework na prática** – Gaste tempo no que é precioso. São Paulo: Leanpub, 2014.

SOUZA, Lucas. **Ruby** – Aprenda a Programar na Linguagem mais divertida. São Paulo: Casa do Código, 2013.

TATE, Bruce A.; HIBBS, Curt. **Ruby on Rails** – Executando. Rio de Janeiro: Alta Books, 2006.