UNIVERSIDADE CATÓLICA DE PERNAMBUCO
CENTRO DE CIÊNCIAS E TECNOLOGIA
TRABALHO DE CONCLUSÃO DE CURSO

Estudo e Avaliação de Boas Práticas de Engenharia de Software para o Desenvolvimento de um Sistema Web Visando a Melhoria da Produtividade e Manutenibilidade

por

Thiago Barros de Oliveira

Recife, junho de 2016.

UNIVERSIDADE CATÓLICA DE PERNAMBUCO
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO

Estudo e Avaliação de Boas Práticas de Engenharia de Software para o Desenvolvimento de um Sistema Web Visando a Melhoria da Produtividade e Manutenibilidade

por

Thiago Barros de Oliveira

Especificação do Projeto do TCC, apresentado ao curso de Ciência da Computação da Universidade Católica de Pernambuco, como parte dos requisitos necessários à obtenção da nota da disciplina INF1808 – Trabalho de Conclusão de Curso.

ORIENTADOR: Antônio Mendes da Silva Filho

Recife, junho de 2016.

Resumo da Monografia Apresentada ao Curso de Ciência da Computação da Universidade Católica de Pernambuco.

Estudo e Avaliação de Boas Práticas de Engenharia de Software para o Desenvolvimento de um Sistema Web Visando a Melhoria da Produtividade e Manutenibilidade

Thiago Barros de Oliveira

Junho / 2016

Orientador: Antônio Mendes da Silva Filho, Doutor.

Área de Concentração: Engenharia de Software

Palavras-chave: **Arquitetura de Software, Código Limpo, Padrões de Projeto, Frameworks.**

Número de Páginas: 73.

RESUMO: *Softwares* estão em constante evolução, assim, desenvolver uma arquitetura flexível e reutilizável torna-se importante para manter o *software* instável e de fácil manutenção e evolução. Este trabalho visa apresentar boas práticas de Engenharia de Software aplicadas no desenvolvimento de um sistema web responsável por gerenciar recursos corporativos, integrando dados e processos de uma organização.

Abstract of Dissertation Presented to UNICAP.

Study and Evaluation of Good Habits of Software Engineering to the Development of a Web System aiming the Improvement of the productivity and maintainability

Thiago Barros de Oliveira

June / 2016

Supervisor: Antônio Mendes da Silva Filho, Doctor.

Area of Concentration: Software Engineering

Keywords: **Software Architecture, Clean Code, Design Pattern, Frameworks.**

Number of Pages: 73.

ABSTRACT: Software are constantly evolving , thus developing a flexible architecture and reusable It -If Important paragraph Keep Unstable software and Easy Maintenance and Evolution . This work aims to present Best Practice Software Applied not hum Development Engineering web system responsible for managing Corporate Resources , integrating data and processes an Organization.

AGRADECIMENTOS

Primeiramente gostaria de agradecer a Deus, pois sem ele nada disso seria possível. A minha família, muito obrigado pelo apoio dado ao longo de toda a minha formação acadêmica, e que não foi diferente durante este trabalho.

Um agradecimento especial a minha namorada pelo apoio e companheirismo, enfrentando os momentos difíceis e a distância durante boa parte do curso.

Aos meus amigos da turma “Por Hoje” pelo companheirismo e diversão.

Aos meus amigos da empresa Ético Software, que acompanharam de perto este trabalho.

Ao professor Antônio Mendes da Silva Filho pelo apoio no desenvolvimento do presente trabalho.

A UNICAP, instituição responsável por minha formação de bacharel em Ciência da Computação.

LISTA DE ABREVIAÇÕES E SIGLAS

UNICAP – Universidade Católica de Pernambuco

TCC – Trabalho de Conclusão de Curso

ABES – Associação Brasileira das Empresas de Software

TI – Tecnologia da Informação

MVC – *Model-view-controller*

UI – *User Interface*

ORM – *Object-Relational Mapper*

SGBD – Sistema Gerenciador de Banco de Dados

ERP – *Enterprise Resource Planning*

JPA – *Java Persistence API*

OO – Orientação a Objeto

GoF – *Gang of Four*

JSF – *JavaServer Faces*

HTML – *HyperText Markup Language*

AJAX – *Asynchronous Javascript and XML*

SQL - *Structured Query Language*

XML - *Extensible Markup Language*

CDI - *Context and Dependency Injection*

CSS - *Cascading Style Sheets*

IDE - *Integrated Development Environment*

HTTP - *Hypertext Transfer Protocol*

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

Escrever código de forma ilegível e sem critérios de codificação pode funcionar, mais acaba por criar uma dívida técnica[1] caso o projeto seja de médio ou grande porte, onde existe ao menos uma equipe de desenvolvedores modificando constantemente o código fonte da aplicação. O acúmulo de diversas dívidas técnicas em vários pontos do código resulta na cobrança de juros altos que são pagos com a perda constante de produtividade.

Saber codificar de forma produtiva e limpa requer experiência, é preciso conhecer os problemas que podem aparecer durante a modelagem de um sistema e saber quais soluções podem ser adotadas para melhor atender aos requisitos. Por exemplo, aplicar padrões de projetos na arquitetura do sistema é uma alternativa que pode ser empregada em problemas recorrentes e já solucionados com sucesso por desenvolvedores em projetos passados. O uso de padrões de projetos possibilita uma maneira de aprender com experiências de outros desenvolvedores e absorver o conhecimento que eles levaram diversos anos para adquirir, economizando assim tempo e evitando ter que pensar em uma solução para um problema comum (GUERRA, 2013).

É muito importante que a maior parte do desenvolvimento esteja voltado para a regra de negócio do sistema, que são tarefas mais complexos e que realmente necessitam de mais atenção. Tarefas clássicas como consultar em banco de dados, tratamento de transação e requisições web entre outras, são problemas que devem ser minimizados com a adoção de bibliotecas e *frameworks* que visam uma padronização no desenvolvimento e a velocidade na produção do *software*(programas de computador), além de outras vantagens como manutenção e segurança.

Este trabalho visa investigar técnicas de desenvolvimento de software assim como frameworks que podem melhorar a produtividade e manutenibilidade, bem como prover um código mais limpo e menos sujeito a modificações sem que estas gerem problemas em outras partes do sistema.

[1] http://martinfowler.com/bliki/TechnicalDebt.html

1.1 MOTIVAÇÃO

Segundo o estudo sobre Mercado Brasileiro de Software feito pela Associação Brasileira das Empresas de Software(ABES) publicado em Junho de 2015 foram identificadas cerca de 12.660 empresas dedicadas ao desenvolvimento, produção, distribuição de software e de prestação de serviços no mercado nacional, sendo 55% delas com atividade principalmente voltada para o desenvolvimento e produção de *software* ou prestação de serviços[2]. Em 2014, a utilização de *software* desenvolvido no país cresceu 19,1%, índice bastante superior aos 11,5% de crescimento identificado no uso de programas de computador desenvolvidos no exterior, reforçando a tendência de crescimento que vem sendo apontada desde 2004.

Observa-se no cenário Brasileiro de Tecnologia da Informação(TI) que a demanda por *softwares* vem crescendo com o passar dos anos, tendo em vista que as empresas não sobrevivem nos dias atuais sem o uso de Tecnologia da Informação. Assim pode-se observar um mercado bastante competitivo onde as empresas de *software* buscam continuamente por melhorias nos métodos de desenvolvimento visando a entrega do produto com qualidade o mais rápido possível.

É investido bastante tempo em especificação de requisitos e planejamento da arquitetura para que só depois as tarefas sejam divididas entre a equipe de desenvolvimento, cuja a função seria transformar os requisitos em código. Em se tratando de métodos ágeis, onde o foco está na entrega de funcionalidades que possam ser rapidamente colocadas em produção a situação melhora um pouco, mas não há como negar que diferentes programadores deixem de fazer alterações e extensões continuamente no código do projeto.

Software evoluem, são constantemente modificados, novos recursos podem ser implementados ou problemas corrigidos, outros são criados e aparecem repentinamente, bem como novas exigências podem ser feitas por parte do cliente. Assim o código pode torna-se complexo e diferente da sua versão inicial.

> "*As pesquisas em geral concordam que a manutenção de software ocupa uma proporção maior dos orçamentos de TI que o desenvolvimento (a*

[2] http://central.abessoftware.com.br/Content/UploadedFiles/Arquivos/Dados%202011/ABES-Publicacao-Mercado-2015-digital.pdf

> *manutenção detém, aproximadamente, dois terços do orçamento, contra um terço para desenvolvimento).*"
>
> (SOMMERVILLE, 2011).
>
> "*De acordo com pesquisa do Standish Group, praticamente um em cada quatro projetos falha em sua execução. Esse fiasco de 24% dos projetos está relacionado a problemas em executá-los dentro do prazo estipulado, dentro do orçamento previsto ou simplesmente por falta de eficiência na gestão para fazer com que os resultados esperados sejam atingidos.*"
>
> (MELO, 2012)
>
> "*Ao longo de um ou dois anos, as equipes que trabalharam rapidamente no início de um projeto podem perceber mais tarde que estão indo a passos de tartaruga.*"
>
> (MARTIN, 2009).

Talvez para atender a prazos apertados e muitas vezes com orçamentos limitados não é dado muita importância a qualidade do código fonte do *software* e com passar do tempo mudanças que muitas vezes são consideradas triviais acabam por impactar vários pontos do código, podendo aumentar o grau de complexidade e também o tempo para solução do problema.

Observa-se que em se tratando de desenvolvimento de sistemas, a exigência de maior produtividade pode comprometer a qualidade, clareza e organização do código fonte de um projeto.

1.2 OBJETIVOS

Objetivo Geral:

- Desenvolver um sistema web aplicando técnicas de codificação limpa, padrões de projeto orientado a objetos e *frameworks* que auxiliem na produtividade e possíveis manutenções futuras.

Objetivos Específicos:

- Investigar técnicas de codificação limpa e critérios para uso padrões de projetos e *frameworks* visando prover suporte a reusabilidade de código e produtividade
- Fazer um levantamento dos requisitos do sistema web para um estudo de caso.
- Selecionar os *frameworks*, padrões de projeto, ferramentas e bibliotecas para uso no estudo de caso.
- Desenvolver o *software* proposto para estudo de caso.
- Avaliar os benefícios e limitações do uso de técnicas de codificação limpa, padrões de projetos e uso de *frameworks*.

1.3 METODOLOGIA

Neste trabalho, os exemplos, soluções e estudo de caso foram escritos na plataforma Java, amplamente conhecida e utilizada no mercado atual.

O projeto foi implementado utilizando o Eclipse Java EE Mars 1 juntamente com a linguagem de programação Java na versão 8, sendo utilizado o Tomcat na versão 8 como servidor web para receber as solicitações da aplicação. Todo o projeto foi desenvolvido em um ambiente Windows 7 64bit.

As ferramentas utilizadas para o desenvolvimento do trabalho foram: Microsoft Word, (para a escrita dos relatórios e da monografia), Microsoft Power Point (para a criação dos *slides* para apresentação).

O projeto foi desenvolvido obedecendo às seguintes etapas:

1. Pesquisa bibliográfica;
2. Estudo de fundamentos de orientação a objetos;
3. Estudo sobre arquitetura de *software;*
4. Estudo das técnicas de codificação limpa;

5. Estudo das técnicas de padrões de projeto orientado a objetos;
6. Estudo de *frameworks*;
7. Desenvolvimento de um sistema web utilizando um padrão arquitetural, técnicas de codificação, padrões de projeto, bibliotecas auxiliares e *frameworks* apresentados neste trabalho.
8. Redação da monografia;

1.4 ORGANIZAÇÃO DO TCC

Este TCC está organizado da seguinte forma:

- Capítulo 2: apresenta tópicos relacionados a arquitetura de *software*, orientação a objetos para o desenvolvimento de um *software* flexível, codificação limpa visando um código de fácil entendimento e padrões de projetos orientado a objetos.
- Capítulo 3: apresenta os *frameworks* estudados que são bases para o estudo de caso do Capítulo 4.
- Capítulo 4: apresenta a estrutura organizacional do projeto prático e explica as funcionalidades oferecidas pelo sistema.
- Capítulo 5: apresenta conclusões, bem como discute limitações do trabalho e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo é dividido em quatro partes, onde a primeira contém conceitos sobre arquitetura de *software*; já a segunda parte apresenta tópicos importantes do paradigma de orientação a objetos que proporcione uma arquitetura que facilite a produção de um código flexível; a terceira parte descreve técnicas de codificação limpa que ajudam a tornar o código fonte organizado e legível e por último são apresentados conceitos de padrões de projeto.

2.1 ARQUITETURA DE SOFTWARE

Uma das certezas no processo de desenvolvimento de *software* é a mudança. *Software*, de uma maneira geral, se encontra em constante mudança. Essas podem ocorrer por correção de erros existentes no código fonte, ou para o agregar novas funcionalidades.

A medida que o *software* cresce, sua complexidade aumenta, e a arquitetura de *software* desempenha um papel fundamental para gerenciar a complexidade inerente ao *software*.

> "*Um bom design de software visa a uma arquitetura flexível que permita futuras alterações, facilite a produção de código organizado e legível, maximizando o seu reaproveitamento.*"
>
> (SILVEIRA et al., 2012).

2.1.1 Definição

> "*A arquitetura de software de um programa ou sistema computacional é a estrutura ou estruturas do sistema, que abrange os componentes de software, as propriedades externamente visíveis desses componentes e as relações entre eles.*"
>
> (BASS, CLEMENTS e KAZMAN, 2003).

"*Arquitetura é o termo que muitas pessoas tentam definir com pouca concordância. Há dois elementos comuns: um deles trata do mais alto nível de quebra de um sistema em partes; o outro trata de uma decisão difícil de ser mudada.*"

(FOWLER, 2002).

"*A arquitetura de software é um conjunto de declarações, que descreve os componentes de software e atribui funcionalidades de sistema para cada um deles. Ela descreve a estrutura técnica, limitações e características dos componentes, bem como as interfaces entre eles. A arquitetura é o esqueleto do sistema e, por isso, torna-se o plano de mais alto-nível da construção de cada novo sistema.*"

(KRAFZIG, BANKE e SLAMA, 2004).

Com base nas definições citadas, é possível entender que a arquitetura de *software* define os elementos que irão compor o software e as regras de comunicação entre esses elementos.

2.1.2 Importância da Arquitetura de Software

Na obra de Bass et al. (2003) apontam três razões-chave da importância de arquitetura de *software*:

1. As representações da arquitetura de *software* são um facilitador para a comunicação entre todas as partes interessadas no desenvolvimento de um sistema computacional.

2. A arquitetura evidência decisões de projeto iniciais que terão profundo impacto em todo o trabalho de engenharia de *software* que vem a seguir e, tão importante quanto, no sucesso final do sistema como uma entidade operacional.

3. A arquitetura "constitui um modelo relativamente pequeno e intelectualmente compreensível de como o sistema é estruturado e como seus componentes trabalham em conjunto".

Com base no que foi relatado, a arquitetura de *software* habilita a comunicação entre os interessados sobre como o *software* vai ser construído, fornecendo uma visão geral da organização e relacionamento dos componentes e ressalta decisões de design que têm profundo impacto em todo trabalho.

A definição da arquitetural é uma etapa fundamental de um projeto e, feita corretamente, ajuda a gerenciar os recursos da equipe e mudanças futuras, que é a única certeza dentro de um projeto, assim reduz os riscos associados com a construção do software.

2.1.3 Decisões Arquiteturais

Para decidir qual melhor escolha a ser tomada, é preciso levar em consideração os requisitos que motivam e justificam a escolha da arquitetura.

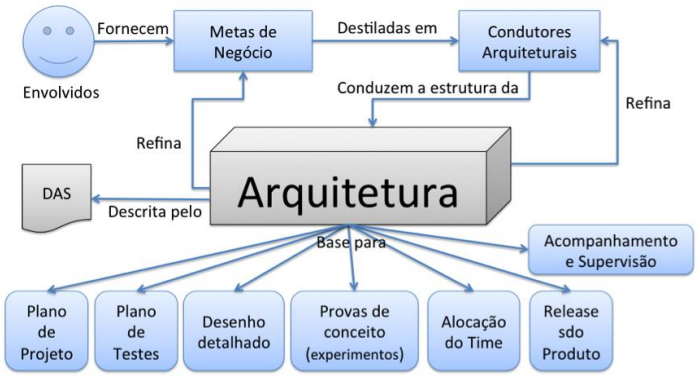

Figura 2. 1: Papel da arquitetura de software no processo de desenvolvimento.[3]

"*Cada visão desenvolvida como parte da descrição arquitetural trata uma necessidade específica do interessado. Para desenvolver cada visão (e a descrição arquitetural como um todo), o arquiteto de sistemas considera uma*

[3] http://www.institutopangea.org/blog/15-a-importancia-da-arquitetura-de-software

variedade de alternativas e, por fim, decide sobre as características de uma arquitetura específica que melhor atendam à necessidade."

(PRESSMAN, 2011).

A Figura 2.1 ilustra o processo em que as partes interessadas fornecem as metas de negócio(necessidades específicas), que seriam os requisitos(itens levantados no processo de levantamento de requisitos), esses por sua vez quando são analisados por parte do arquiteto de *software* que considera uma variedade de alternativas e escolhe a que melhor atende à necessidade do cliente.

Desta forma, não é possível se criar uma arquitetura que atenda toda a gama de tipos de aplicações diferentes.

Assim, o ideal é que a arquitetura de um *software* seja definida caso a caso, embora se saiba que existem empresas que por questão de tempo, ou por não terem profissionais adequados para fazer uma análise detalhada de cada problema acaba definindo de início as decisões arquiteturais padrões.

2.2 TÓPICOS DE ORIENTAÇÃO A OBJETOS

Construir um *software* que realmente seja orientado a objetos não é uma tarefa fácil. Utilizar orientação a objeto (OO) de verdade, exige maturidade, prática e entender seus princípios, para que se possa utilizar realmente todos os benefícios da orientação a objetos.

O que geralmente acontece, é que os desenvolvedores se preocupam somente no que é necessário para fazer o *software* funcionar, e não considera questões como manutenção ou construir um código que seja fácil de modificar e que siga realmente a orientação a objetos de fato.

A seguir cito tópicos importantes que auxiliam a tornar o código flexível e que devem ser levados em consideração em um projeto orientado a objetos.

2.2.1 Programação voltada a abstração, não a implementação

No momento de criação de uma nova classe, deve-se pensar em o que ela irá fazer(contrato), ao invés de se pensar em como ela irá fazer(implementação), desacoplando totalmente a interface da classe de sua implementação.[4]

O uso de interface promove o polimorfismo além de ajudar no desacoplamento de qualquer código que as utilize, pois interfaces são apenas contratos, ou seja, uma série de métodos sem nenhuma implementação.

A idéia de usar interface é possibilitar mudar a implementação sem provocar erros de compilação em diversos pontos do código, ou seja, sem afetar a abstração, tornando o código flexível. Para um melhor entendimento, segue a Figura 2.2.

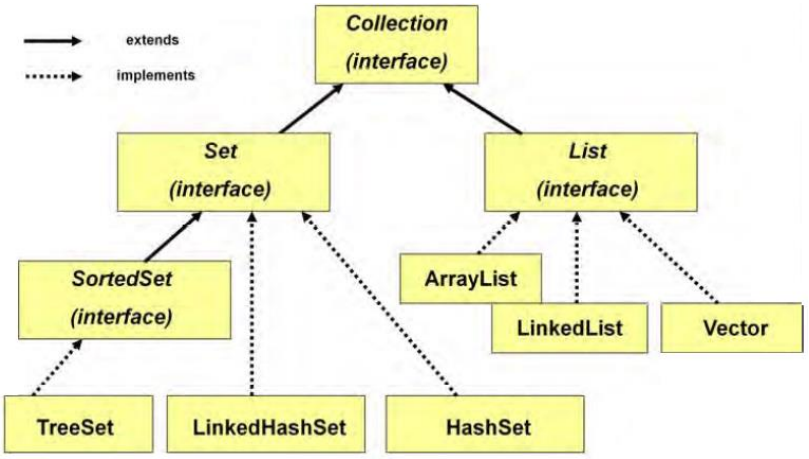

Figura 2. 2: Familia de classes da API Collection em Java[5]

Na Figura 2.2 é apresentado as classes e interfaces responsáveis por manipulações de coleções em java.

[4] http://blog.raphaelivan.com.br/programe-voltado-para-a-interface-e-n%C3%A3o-para-a-implementa%C3%A7%C3%A3o.html
[5] http://cdn.crunchify.com/wp-content/uploads/2013/04/Java-Collection-Framework-Crunchify.png

Escolher a implementação(classe) correta para cada caso é uma tarefa complicada. Cada uma das classes existe para resolver um tipo de problema específico.

Frameworks e bibliotecas utilizadas no desenvolvimento sempre tiram proveito do uso de interfaces. É o caso do Hibernate, que será citado no capítulo relacionado a *frameworks*, nele, qualquer consulta que retorne uma lista tem como retorno um objeto do tipo *List*, deixando claro ao usuário que a ordem dos elementos é importante e possibilitando todos as funcionalidades que uma *List* oferece, como navegação indexada (SILVEIRA et al., 2012).

2.2.2 Evite herança, favoreça composição

Quando se desenvolve *software*, diversos são os momentos em que pode ser feito uso de herança para implementar funcionalidades de maneira rápida. Mas, encadear comportamentos através de herança de classes apresenta diversos problemas de acoplamento e dificulta a customização de um processo.

"*O maior problema com a herança é que acoplamos a implementação da classe mãe muito precocemente, criando a necessidade de a classe filha conhecer muito bem o código interno da classe mãe, o que é visto como uma quebra de encapsulamento.*"

(SILVEIRA et al., 2012).

No obra de Joshua Bloch (2008) são citados vários itens importantes na construção de um projeto OO, o item 14 diz, "*prefira composição em vez de herança*". Joshua Bloch afirma que herança quebra o encapsulamento. Fowler (2006) em post publicado *Designed Inheritance*[6] segue a mesma linha de pensamento e desencoraja o uso de herança.

[6] http://martinfowler.com/bliki/DesignedInheritance.html

2.2.3 Baixo acoplamento, alta coesão

Coesão e acoplamento, são princípios de engenharia de *software* que embora sejam simples, muitas das vezes são ignorados. Entender os conceitos atrelados a acoplamento e coesão influência no bom design de um *software*.

Martin (2009), defende em sua obra que se uma classe tem mais de um motivo para ser alterada ela já está ferindo o princípio da responsabilidade única, sendo assim, com baixa coesão. Ainda para o autor, as classes devem ter apenas uma responsabilidade, realizar somente ela, e realizá-la bem.

> "*Ao desenhar pequenas classes que, juntas, realizam uma tarefa maior, o programador ganha em simplicidade e modularidade.*"
>
> (SILVEIRA et al., 2012).

É mais simples desenvolver um *software* composto por classes que contenham apenas uma responsabilidade bem definida, pois as alterações de um comportamento é feito em um único local do código (SILVEIRA et al, 2012).

Códigos coesos são aqueles de relação forte, onde seus membros estão intimamente ligados e estão ali por um objetivo comum. Membros que não são absolutamente necessários para aquele módulo não devem estar presentes em códigos coesos.

Já o acoplamento está atrelado a o quanto uma classe depende da outra para funcionar, e quanto maior é esta dependência entre ambas diz-se que estão fortemente acopladas, originando problemas.

No desenvolvimento de um *software* vários componentes(classes, objetos) devem se comunicar, desta forma existindo uma ligação. Com o objetivo de facilitar a manutenção a longo prazo, deve-se lutar para que as ligações entre os componentes seja a menor e mais simples possível (SILVEIRA et al., 2012).

> "*Encapsulamento e bom uso de interfaces são essenciais para diminuir o acoplamento entre componentes.*"
>
> (SILVEIRA et al., 2012).

2.3 CODIFICAÇÃO LIMPA

É através do código que se expressa todos os fluxos de comportamento do *software*, é com o código que se informa para a máquina como determinado processo deve ser executado. É possível utilizar linguagens de mais alto nível ou bibliotecas que abstraiam as partes técnicas mas tem-se que codificar a regra de negócio particular de cada sistema de forma minuciosa, exata, com todos os detalhes para que a máquina possa entender e executar a tarefa.

> "*Escrever código que possa ser entendido e executado por uma máquina é fácil, todavia escrever código que possa ser lido e entendido por humanos é bem mais complexo.*"
>
> (KERIEVSKY, 2008).

2.3.1 Definição de código limpo

Martin (2009) em sua obra, relata que realizou diversas entrevistas com programadores bem conhecidos e com muita experiência, dentre as várias citações, destaco a de Booch que diz, "*Um código limpo é simples e direto. Ele é tão bem legível quanto uma prosa bem escrita. Ele jamais torna confuso o objetivo do desenvolvedor, em vez disso, ele está repleto de abstrações claras e linhas de controle objetivas.*".

Assim um código limpo deve ser:

- Simples – Deve conter apenas a complexidade necessária para resolver o problema;

- Direto – Deve está claro o comportamento do código.
- Eficiente – Deve ser escrito de forma a resolver o problema da forma mais correta com as ferramentas disponíveis pela linguagem.
- Sem duplicidade – Se já existe um código que realiza a operação, ele deve ser invocado.
- Elegante – O código deve conter nomes representativos, além de estar bem identado.
- Feito com cuidado – A codificação deve ser feita com cuidado, sem pressa.

2.3.2 Por que usar técnicas de codificação limpa?

Um código ruim pode atrasar o desenvolvimento, a produtividade da equipe é diretamente afetada. Mudança alguma é trivial, cada alteração requer uma análise profunda em vários pontos do sistema (MARTIN, 2009).

É possível identificar um código ruim quando se tenta modificar o comportamento de um determinado módulo, seja para estender uma determinada funcionalidade ou para corrigir um determinado bug e não se localiza o ponto onde deve ser feito a alteração, ou quando é feita a alteração e se observa comportamentos ainda mais estranhos em pontos totalmente diferentes do mesmo código.

A Figura 2.3 mostra que no início de um projeto a produção é rápida e sem grandes problemas já que não existe muito código desenvolvido. Com o passar do tempo e a má prática de codificação observa-se um declínio gradativo na produção, o desenvolvimento já não se mostra tão rápido e isso se agrava com o passar do tempo.

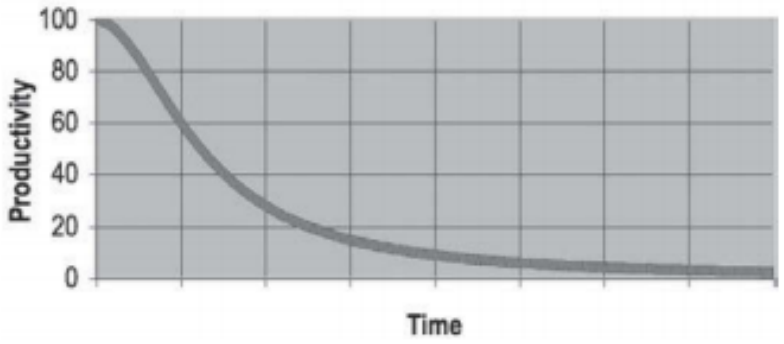

Figura 2. 3: Produtividade x Tempo (MARTIN, 2009)

"*Dedicar tempo para limpar seu código não é apenas eficaz em termos de custo, mas uma questão de sobrevivência profissional.*"

(MARTIN, 2009).

Ainda, com a codificação limpa, as perdas geradas por alterações na equipe são reduzidas. Outro programador irá conseguir entendê-lo facilmente e, o ritmo da equipe tem uma maior chance de se manter estável.

É importante obter um código bem organizado para que a produtividade não seja afetada e assim a implementação dos requisitos seja entregue no menor tempo possível, evitando ultrapassar prazos o que diminui a credibilidade da empresa.

2.3.3 Técnicas de codificação

A seguir, em resumo, cito técnicas de codificação limpa explicadas por Martin (2009) em sua obra, trata-se de levantamentos de pontos importantes que devem ser adotados no desenvolvimento de um projeto:

- **Nomes significativos**: Nomes de classes, variáveis, métodos, entre outros, devem ser significativos, indicando claramente o que faz ou o que representa.

A intenção da existência do componente deve ser visível através do seu nome. Deve-se criar nomes pronunciáveis afim de facilitar a comunicação com os demais membros da equipe. Devemos evitar siglas.

- **Funções:** Funções devem ser pequenas e deve haver apenas um nível de abstração por função, ou seja, elas devem realizar apenas uma tarefa.

  Novamente, utilize um nome que descreva bem o que a função realiza. Utilize o menor número de argumentos possível, Martin (2009) recomenda o máximo três parâmetros.

  Cuidado com "efeitos colaterais", funções não podem fazer nada "escondido" (nada além do que se propõe a fazer), isso fere o principio de responsabilidade única.

  Martin (2009) também recomenda o uso de exceções ao invés de códigos de erro como retorno de funções.

- **Comentários:** Deve-se saber que se comentarmos um código é porque ele deve estar complexo demais, com isso devemos simplifica-lo e não comentá-lo.

  Procure explicar o que o código faz em código e não em comentário.

  Crie nomes de métodos e de variáveis informativos, ao invés de explicar com um comentário o que um método com um nome ruim faz.

  Não comente código que não será mais usado, simplesmente remova-o. Esta é uma forma de não poluir o código e ferramentas de controle de versão existem também para auxiliar a buscas por código de versões antigas, o que não justifica comentar código inútil.

- **Tratamento de erro**: Deve-se usar exceções ao invés de códigos de erro. Recomenda-se que essas exceções não sejam do tipo checadas.

  As mensagens de erro em exceções devem ser de caráter informativo.

  Não é aconselhável retornar null, isso pode causar uma famosa NullPointerException (no caso do Java).

- **Testes**: É recomendado a adoção das três leis de teste.

    1. Não escrever nenhum código até ter escrito um teste que detecte uma possível falha.
    2. Não escrever mais testes de unidade do que o suficiente para detectar a falha.
    3. Não escrever mais código do que o suficiente para passar nos testes.

- **Classes**: Devem ser pequenas e seguir o princípio da responsabilidade única. Devem ser coesas, com isso, resultando em classes pequenas. Classes devem ser criadas visando a mudança, então a programação deve ser orientado à interface.

Outras considerações relatadas por Martin (2009) que são de grande importância são:

1. Todos os testes devem estar passando;
2. Refatoração deve ser feita constantemente, visando a melhoria contínua. Na obra de Fowler (2004) é relatado um guia com vários procedimentos a serem adotados para a refatoração de código;
3. Código duplicado deve ser evitado a todo custo;

2.3.4 Identificando código de qualidade

Com base no que já foi relatado no presente trabalho, um código de qualidade deve estar modularizado, cada classe deve ter sua responsabilidade, e as relações entre elas devem estar bem definidas.

Com o objetivo de facilitar a análise de um código com relação a sua qualidade Martin em 2000 ajudou a popularizar o acrônimo S.O.L.I.D.,

introduzido por Michael Feathers, após observar que os cinco princípios poderiam se encaixar nesta palavra.[7]

- **Single Responsibility Principle (SRP), ou, Princípio da Responsabilidade Única:** Uma classe deve ter um, e somente um, motivo para mudar(ser alterada).
- **Open Closed Principle (OCP), ou Princípio do Aberto Fechado:** Uma classe deve estar aberta para extensão, mas fechada para alteração
- **Liskov Substitution Principle (LSP), ou Princípio da Substituição de Liskov:** Deve ser possível substituir uma classe base por uma classe derivada em qualquer ponto do código.
- **Interface Segregation Principle (ISP), ou Princípio da Segregação de Interfaces:** Preferir interfaces específicas do que ter interfaces genéricas.
- **Dependency Inversion Principle (DIP), ou Princípio da Inversão de Dependências:** Dependa de uma abstração e não de uma implementação.

2.4 PADRÕES DE PROJETO DE SOFTWARE

Para se construir um *software* orientado a objetos de sucesso não basta apenas entender o respectivo paradigma. Saber o que é uma classe, um objeto, saber como se implementa uma herança ou um polimorfismo significa conhecer as ferramentas disponíveis para se programar orientado a objetos, é preciso saber onde e como usar essas ferramentas (GUERRA, 2013).

"*Projetar software orientado a objetos é difícil, mas projetar software reutilizável orientado a objetos é ainda mais complicado. Você deve identificar objetos pertinentes, fatorá-los em classes no nível correto de granularidade,*

[7] http://eduardopires.net.br/2013/04/orientacao-a-objeto-solid/

definir as interfaces das classes, as hierarquias de herança e estabelecer as relações-chave entre eles."

(GAMMA et al., 1995).

No desenvolvimento, muitas vezes, depara-se com problemas que outras pessoas já resolveram, problemas que ocorrem com frequência e que são, de certa forma, comuns.

O tempo desperdiçado para a solução do problema poderia ser reduzido caso o desenvolvedor tivesse conhecimento sobre os padrões de projeto existentes. Com a utilização de padrões, as soluções para esses problemas se dá de forma mais rápida.

2.4.1 Definição de Padrões de Projeto

O termo padrão de projeto é originado do livro "A Patterns Language – Towns, Buildings, Construction" (Uma Linguagem de Padrões – Cidades, Edificações, Construções) de Christopher Alexander, na área de arquitetura e cidades e construções. Na obra, Alexander descreve padrões em construções e cidades, porém pode-se notar uma semelhança com padrões de projeto orientados a objetos.[8]

"*Cada padrão descreve um problema no nosso ambiente e o cerne da sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira.*"

(ALEXANDER, 1977).

Após o lançamento da obra de Alexander diversos autores publicaram seus trabalhos a respeito de padrões e em diferentes áreas. Já na área de *software*, foram propostos padrões para diversas fases do processo de

[8] http://tcc.ecomp.poli.br/20062/Monografia_TiagoMoraes.pdf

desenvolvimento: padrões de requisitos, padrões de análise, padrões de arquitetura, padrões de projeto, entre outros.

A disseminação dos padrões na área de desenvolvimento de software iniciou-se com o livro que é considerado referência no assunto "*Design Patterns: Elements of Reusable Object-Oriented Software*", este livro também é conhecido como GoF, um acrônimo de Gang of Four, uma referência aos seus quatro autores, nele é descrito diversos padrões que são utilizados até os dias atuais.

"*Um padrão descreve um conjunto composto por um contexto, um problema, e uma solução. Em outras palavras, pode-se descrever um padrão como uma solução para um determinado problema em um contexto. Porém um padrão não descreve qualquer solução, mas uma solução que já tenha sido utilizada com sucesso em mais de um contexto.*"

(GAMMA et al., 1995).

"*Um padrão é uma ideia que foi útil em um contexto prático e que provavelmente será útil em outros.*"

(FOWLER, 1997).

Assim, um padrão é apresentado como um guia, pois não descreve a solução final para os problemas, mas sim, um ponto de partida que pode ser ajustado de acordo com a necessidade do desenvolvedor para resolver um problema específico.

Pode-se afirmar que um padrão, de modo geral, descreve um problema recorrente e apresenta uma solução elegante já utilizada em vários projetos e que funcionaram perfeitamente.

2.4.2 Por que usar padrões de projeto?

A aplicação de padrões de projeto possibilita várias vantagens no processo de desenvolvimento de *software* orientado a objetos.

Exemplos de benefícios são:

1. A reutilização de uma solução, ao invés do desperdício de tempo em busca do que já existe.
2. Aprimora a comunicação entre os desenvolvedores da equipe já que cada padrão tem um nome que remete a sua solução.
3. Com uso de padrões de projeto, mudanças torna-se menos trabalhosas devido ao uso da orientação a objetos.
4. Reduz a complexidade do código além de uma solução elegante.

2.4.3 Elementos de um padrão de projeto

Para Gamma et al. (1995), um padrão tem quatro elementos essenciais:

1. **Nome:** Ter um nome definido facilita na comunicação com os demais desenvolvedores da equipe, o nome pode ser usado para descrever um problema de projeto ou uma solução para um determinado problema. Normalmente o nome é composto por uma ou duas palavras.
2. **Problema:** Todo padrão surge a partir de um problema.
3. **Solução:** Trata-se dos componentes que se relacionam com o objetivo de solucionar o problema. Essa solução não é descrita de forma exata, pois um padrão é um conceito que deve ser adaptado a cada caso.
4. **Consequências:** São os resultados da aplicação do padrão, suas vantagens e desvantagens.

2.4.4 Catálogo de Padrões de Projeto

Na obra de Gamma et al. (1995), devido a existência de vários padrões, houve a necessidade de organizá-los em um catálogo. Esse catálogo classifica os padrões de acordo com seu propósito e seu escopo.

A Figura 2.4 é uma representação do catálogo dos padrões de projeto.

		Propósito		
		De criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Method (112)	Adapter (class) (140)	Interpreter (231) Template Method (301)
	Objeto	Abstract Factory (95) Builder (104) Prototype (121) Singleton (130)	Adapter (object) (140) Bridge (151) Composite (160) Decorator (170) Façade (179) Flyweight (187) Proxy (198)	Chain of Responsibility (212) Command (222) Iterator (244) Mediator (257) Memento (266) Observer (274) State (284) Strategy (292) Visitor (305)

Figura 2. 4: Catálogo de Padrões de Projeto para Gamma et al.(1995)

Na Figura 2.4 observa-se que existem três finalidades que são a base para a separação dos padrões em categorias distintas. Os padrões de criação são utilizados para situações de criação de objetos. Já os padrões estruturais lidam com a composição de classes e objetos. Os padrões comportamentais tentam definir como as classes e objetos interagem e distribuem responsabilidades entre si. (GAMMA et al. 1995).

Com relação ao escopo pode-se ter padrões que lidam com classes e subclasses onde os relacionamentos são fixados em tempo de compilação, e padrões que lidam com objetos, onde podem ser mudados em tempo de execução, tornando a solução mais dinâmica. (GAMMA et al. 1995).

2.4.5 Descrevendo o padrão Template Method

Todos os padrões presentes na Figura 2.4 são descritos na obra de Gamma et al. (1995), A descrição do padrão *Template Method*, utilizada neste trabalho, compreende:

- Intenção: Definir uma template padrão de um algoritmo postergando passos chaves para as subclasses. Trata-se de um modelo de algoritmo que possui partes fixas e outras partes variáveis dependendo da subclasse.

- Motivação: Quando se conhece o algoritmo porém os detalhes depende do contexto.

  Considere um modelo de relatório. Todo relatório tem um título e os dados de exibição, mas só no momento de execução que é possível saber qual o relatório solicitado, assim a subclasse do relatório específico define qual o título do relatório assim como os dados que serão exibidos.

- Aplicabilidade: Deve-se usar este padrão quando:

  - ✓ Tem-se um algoritmo em que partes são invariáveis e outra dependem do contexto.

  - ✓ Quando se observa comportamentos comuns em classes do mesmo contexto.

- Estrutura: Na Figura 2.5 é possível observar uma classe abstrata *(AbstractClass)* que define um método padrão *(TemplateMethod)* que realiza chamada para métodos abstratos que deve ser definidos na classe filha *(ConcreteClass).*

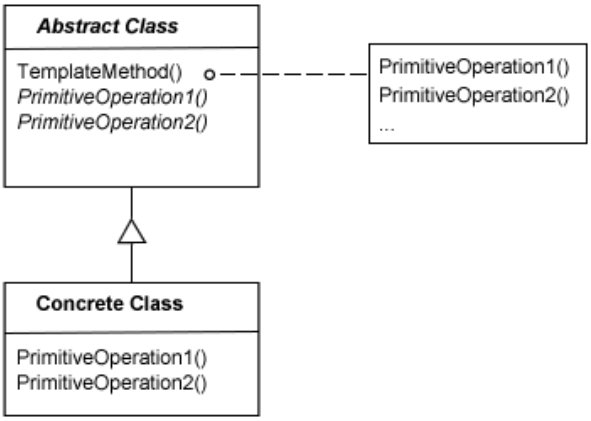

Figura 2. 5: Representação da estrutura do Template Method (GAMMA et al., 1995)

- Participantes:
  - ✓ AbstractClass: Define o algoritmo abstrato.
  - ✓ ConcreteClass: Implementa os passos específicos do algoritmo base de acordo com seu contexto.
- Colaborações: ConcreteClass depende de AbstractClass para implementar os passos específicos do algoritmo.
- Consequências:
  - ✓ Reuso de código.
  - ✓ Inversão de controle, já que os métodos da classe mãe chama as operações definidas na subclasse.
- Implementação:
  - ✓ Os métodos definidos nas subclasses podem ser definidos como protegidos(modificador de acesso *protected*), assim assegura que somente sejam chamados pelo método de template.
  - ✓ Deve garantir que o comportamento varie dependente do objeto manipulado.
- Exemplo de código:

```
public abstract class Relatorio {

    public abstract String getNomeTitulo();
    public abstract List<Registro> getDados();
```

```
        public void gerarRelatorio(){
                String nomeTitulo = getNomeTitulo();
                List<Registro> registros = getDados();

                System.out.println("Título: " + nomeTitulo);

                for(Registro registro : registros) {
                        System.out.println(registro);
                }
        }
}

public class VendasRelatorio extends Relatorio {
        public String getNomeTitulo(){
                return "Vendas";
        }
        public List<Registro> getDados(){
                List<Registro> retorno = new ArrayList<>();
         retorno.add(new Registro("Venda de Geladeira"));
                retorno.add(new Registro("Venda de Fogão"));
        }
}

public class FuncionarioRelatorio extends Relatorio {
        public String getNomeTitulo(){
                return "Funcionários"
         }
        public List<Registro> getDados(){
                List<Registro> retorno = new ArrayList<>();
                retorno.add(new Registro("João da Silva"));
                retorno.add(new Registro("Marcio Castilho"));
        }

}
```

- Uso conhecido: No java, a família da classe Servlet adota o *Template Method* em sua implementação.
- Padrões relacionados: O padrão *Factory Methods* são frequentemente chamados por métodos-template. Em *Template Method* é utilizado herança para variar partes de um algoritmo, já no padrão *Strategy* é utilizado a delegação.

Porém, como já foi relatado, o uso de herança possui limitações. Em java uma classe filha só pode herdar comportamentos de apenas uma classe mãe, isso significa que a classe filha não poderá herdar de nenhuma outra classe. Outra questão é o acoplamento forte que a herança proporciona.

> "*Ao utilizar um padrão, é preciso avaliar os requisitos de sua aplicação e quais consequências pesam mais ou menos. A partir dessas informações é possível decidir se seu uso será ou não adequado.*"
>
> (GUERRA, 2013).

3 FRAMEWORKS

Nesta seção é apresentado os motivos de se adotar frameworks para auxiliar no desenvolvimento de *softwares*. Adicionalmente, descreve-se o funcionamento de *frameworks* utilizados no projeto prático deste trabalho.

3.1 INTRODUÇÃO

Analisando *softwares* corporativos é fácil perceber várias funcionalidades em comum. Autenticação de usuário, acesso a banco de dados, tratamento de requisição http, geração de relatórios, envio de e-mail entre outras.

> "*Uma coisa que os melhores projetistas sabem que não devem fazer é resolver cada problema a partir de princípios elementares ou do zero.*"
>
> (GAMMA et al., 1995).

Visando o reuso de código e a melhora da produtividade a utilização de frameworks e bibliotecas são soluções que podem ser utilizadas pelas empresas de tecnologias.

3.2 DEFINIÇÃO DE FRAMEWORK

"*Framework é um conjunto de classes que constitui um projeto abstrato para a solução de uma família de problemas.*"

(FAYAD & JOHNSON, 1999).

"*Framework é uma arquitetura desenvolvida com o objetivo de atingir a máxima reutilização, representada como um conjunto de classes abstratas e concretas, com grande potencial de especialização*"

(MATTSSON, 1996).

"*Framework é um conjunto de classes cooperantes que constroem um projeto reutilizável para uma determinada categoria de software*"

(GAMMA et al., 1995).

"*Framework é uma arquitetura semi-completa que pode ser instanciada para produzir aplicações customizadas, permitindo o reuso de análise, de projeto e de código*"

(BARBOSA, 2001).

Com base nestas definições pode-se afirmar que um *framework* é uma abstração que une códigos comuns entre vários projetos de *software* provendo uma funcionalidade genérica e reutilizável.

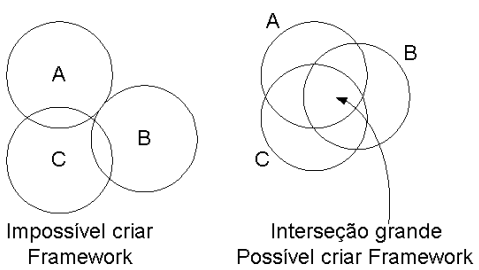

Figura 3. 1: Relação entre diferentes projetos de software[9]

[9] http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/frame/oque.htm

Observando o lado esquerdo da Figura 3.1, pode-se notar três projetos com pouca ou nenhuma intercessão de problemas enfrentados em seus desenvolvimentos. Neste cenário fica difícil e sem motivos reais para a construção ou utilização de um framework em comum a todos.

Já o lado direito da Figura 3.1 a situação é diferente. Nela tem-se três projetos que compartilham problemas semelhantes e é justificado a criação e uso de um determinado frameworks padrão.

Assim, a criação de um frameworks se faz por uma necessidade de resolver problemas específicos em diversos projetos com características em comum.

3.3 POR QUE USAR FRAMEWORKS?

A utilização de *frameworks* é uma estratégia frequente no desenvolvimento de sistemas atuais, por permitir o reuso e a construção de sistemas configuráveis.[10]

Utilizar *frameworks* reduz custos, pois o desenvolvimento já inicia em uma estrutura definida, assim o desenvolvedor pode se concentrar em implementar as regras específicas do negócio.

Um *framework* ainda proporciona uma maior reutilização de códigos e a fatoração de problemas em aspectos comuns a várias aplicações, permite também obter sistemas com códigos com menos defeitos.

3.4 JAVASERVER FACES

As aplicações web são fundamentais para qualquer empresa em um mercado atual. Elas oferecem soluções práticas sem a dependência de *hardware* e de instalação por parte do cliente, possibilitando assim uma maior mobilidade, facilidade de acesso e atualizações de versão do *software*.

Na plataforma java, as tecnologias voltadas para o desenvolvimento de aplicações web têm mudado constantemente, e uma das justificativas para a mudança

[10] http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/frame/porque.htm

seria um desenvolvimento que oferecesse facilidade de manutenção e produção de conteúdo reutilizável.[11]

JavaServer Faces surgiu como uma alternativa ao antigo JSP, (tecnologia até então padrão para implementação de aplicações web em java), e logo ganhou espaço no mercado, se tornando uma das mais utilizadas tecnologias para web hoje em dia.[12]

3.4.1 Definição do JavaServer Faces

JavaServer Faces (JSF) é um *framework* baseado em componentes (campos de formulário) e dirigido a eventos(resultado de uma ação).[13]

Ele permite criar aplicações java para web utilizando componentes visuais pré-prontos de forma que o desenvolvedor não se preocupe com JavaScript e HTML, bastando adicionar os componentes (calendário, tabelas entre outros) que eles serão renderizados e exibidos em formato html.

JSF é que um *framework* responsável por trabalhar com interfaces de usuários para sistema web, colocando componentes em um formulário e ligando os a objetos java, sendo assim ele faz a separação entre a lógica e regras de negócio a navegação e conexões com serviços externos seguindo o modelo *Model-View-Controller* (MVC). Tem como ponto forte a possibilidade de um grande número de componentes e um design bastante flexível por isso esse *framework* vem se acomodando nas novas tecnologias.[14]

3.4.2 Por que usar o JavaServer Faces

O JSF basear-se no padrão MVC, e uma de suas melhores vantagens e a clara separação entre a visualização e regras de negócio.

[11] http://www.cin.ufpe.br/~jvwr/JSF/jsf.pdf
[12] http://www.devmedia.com.br/javaserver-faces-jsf-melhorando-a-qualidade-da-aplicacao-web/30902
[13] https://pt.wikipedia.org/wiki/JavaServer_Faces
[14] http://fabrica.ms.senac.br/2013/06/o-que-e-jsf-java-server-faces/

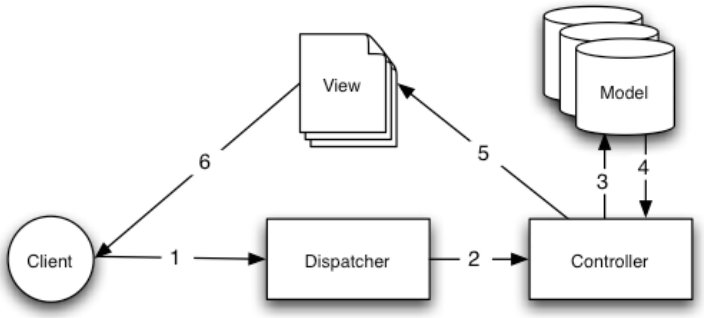

Figura 3. 2: Arquitetura MVC[15]

Na figura 3.2 podemos ter uma melhor noção do comportamento de um sistema em uma arquitetura MVC. O cliente realiza uma solicitação ao controlador que é responsável por realizar toda a regra de negócio e atualização do modelo, após este processo é construído a tela que será exibida para o usuário, já com o modelo atualizado.

O *framework* JSF é responsável por interagir com o usuário, e fornece ferramentas para criar uma apresentação visual. Porém, o escopo de JSF é restringindo à camadas de apresentação. A persistência de banco de dados e outras conexões de back-end estão fora do escopo do *framework*.

Em JSF, o controle é feito através de uma servlet(classe java que trata requisições http) chamada Faces Servlet, opcionalmente, por arquivos XML de configuração e por vários manipuladores de ações e observadores de eventos.

A Faces Servlet recebe as requisições dos usuários na web, redireciona para o modelo e envia uma resposta.

O JSF possui as seguintes partes:

- Um conjunto de componentes de UI (User Interface) predefinidos;
- Um modelo de programação orientada a eventos;
- Um modelo de componentes que permite a desenvolvedores independentes fornecerem componentes adicionais.

[15] http://book.cakephp.org/1.3/pt/_images/basic_mvc.png

JSF é uma especificação java, com isso teve uma aceitação no mercado muito mais fácil e rápida que muitos frameworks escritos por terceiros. A ideia de trabalhar com uma especificação é muito tentadora para diversas empresas, pois garante maior segurança de continuidade da tecnologia (SILVEIRA et al., 2012).

A implementação mais utilizada do JSF e também a implementação de referência, é a Oracle Mojarra que é inclusive a implementação utilizada no projeto prático deste trabalho. Existem outras, como é o caso da MyFaces da Apache Software Foundation.[16]

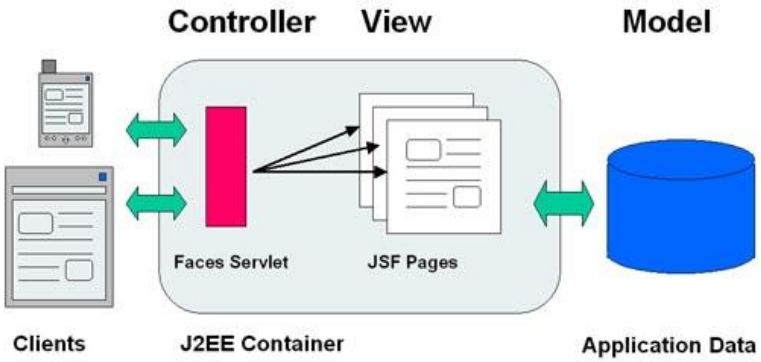

Figura 3. 3: Funcionamento do JSF[17]

Na Figura 3.3 mostra a interação entre cliente e servidor em uma aplicação típica JavaServer Faces.

A solicitação é capturada pelo controlador principal (Faces Servlet), que realiza tratamentos padrões do próprio *framework* e passa para classes do sistema, essa manipula o modelo e ajusta a *view*(páginas JSF) de resposta que será exibida ao usuário.

3.4.3 Ciclo de vida do JSF

[16] http://www.rponte.com.br/2008/02/18/qual-implementacao-jsf-voce-usa/
[17] http://luissoares.com/wp-content/uploads/2014/05/image0021.jpg

O JSF funciona respeitado um poderoso e complexo ciclo de vida dos componentes e requisições, possibilitando o desenvolvedor a acompanhar as diversas fases do fluxo interno de sua execução.[18]

Fases do ciclo de vida:

- **Restaurar Visão (Restore View):** A fase de restauração da visão recupera a hierarquia de componentes para a página solicitada, se ela foi exibida anteriormente, ou constrói uma nova hierarquia de componentes, se for a primeira exibição.
- **Aplicar Valores da Requisição (Apply Request Values):** Nesta fase, cada componente da hierarquia de componentes criada na fase anterior tem a chance de atualizar seu próprio estado com informações que vieram da requisição.
- **Validações de Processo (Process Validations):** Os valores submetidos são convertidos em tipos específicos e anexados aos componentes.
- **Atualizações de Valores do Modelo (Update Model Values):** Durante esta fase, os valores anexados (conhecidos como valores locais) aos componentes são atualizados nos objetos do modelo de dados.
- **Chamada da Aplicação (Invoke Application):** Os eventos que originaram o envio do formulário ao servidor são executados. Por exemplo, ao clicar em um botão para submeter um cadastro, a programação da ação deste botão deve ser executada.
- **Renderização da Resposta (Render Response):** Gera a saída com todos os componentes nos seus estados atuais e envia para o cliente.

Na figura 3.4 temos uma visão de como se comporta o fluxo de uma requisição em um projeto JSF.

[18] http://www.devmedia.com.br/ciclo-de-vida-do-javaserver-faces-jsf/27893

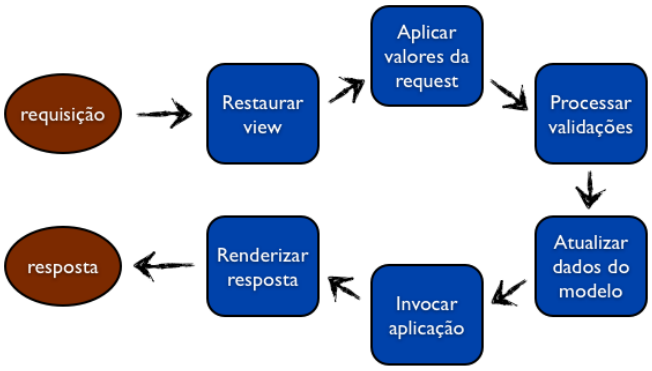

Figura 3. 4: Ciclo de vida JSF[19]

3.4.4 Componentes padrões

A Figura 3.5. mostra componentes padrões oferecidos pela implementação padrão do JSF

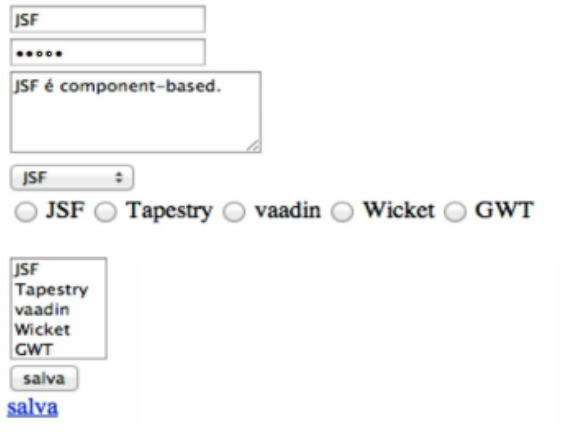

Figura 3. 5: Conjunto de componentes padrões do JSF Mojarra[20]

Pode-se observar que são componentes simplificados, não existe componentes sofisticados dentro da especificação, isso é proposital. Uma especificação tem que ser estável e as possibilidades das interfaces com o usuário crescem muito rapidamente, com isso a especificação só cobre o básico deixando que outros projetos implementem novos componentes.

[19] http://arquivo.devmedia.com.br/REVISTAS/java/imagens/119/2/1.png
[20] https://www.caelum.com.br/apostila-java-testes-jsf-web-services-design-patterns/introducao-ao-jsf-e-primefaces/#7-2-caracteristicas-do-jsf

Existem várias bibliotecas do JSF que oferecem componentes das diferentes formas e propósitos distintos. Exemplos dessas bibliotecas são PrimeFaces[21], RichFaces[22] e IceFaces[23].

Para a definição da interface do projeto prático neste trabalho, é feito uso do Oracle Mojarra com PrimeFaces, uma combinação muito comum no mercado.

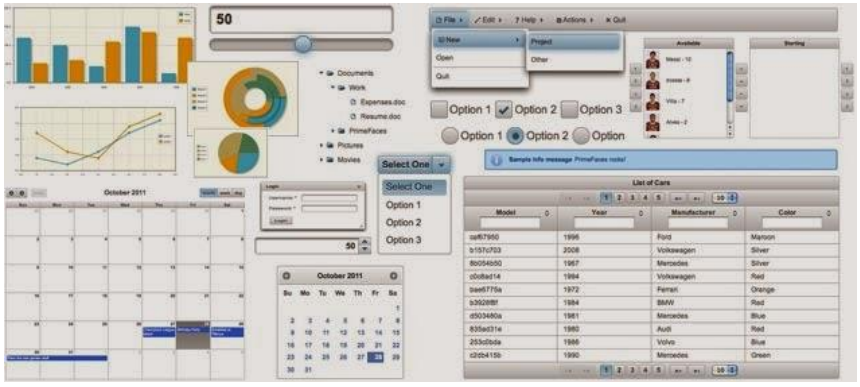

Figura 3. 6: Exemplos de componentes Primefaces[24]

3.4.5 Serviços oferecidos pelo JSF

Os serviços mais importantes oferecidos pelo JSF são:

- **Arquitetura *Model-View-Controller*:** A implementação JSF opera como o controlador que é acionado a partir de ações do usuário, processando eventos de ação e direcionando-os para o código que atualiza o modelo ou a visualização.

```
<h:form>
       Login: <h:inputText value="#{autenticador.login}" />
       </br>
       Senha: <h:inputSecret value="#{autenticador.senha}">
       </br>
       <h:commandButton value="entrar"
action="#{autenticador.logar}"/>
       </br>
       <h:link value="Cadastre-se" outcome="cadastro" />
</h:form>
```

Exemplo HTML de tela de login

[21] http://www.primefaces.org/
[22] http://richfaces.jboss.org/
[23] http://www.icesoft.org/java/projects/ICEfaces/overview.jsf
[24] http://4.bp.blogspot.com/-8jQliWtKvq4/UwtQmTOjMTI/AAAAAAAAAlI/1wZSkFN8s-E/s1600/Primfaces-compoents.jpg

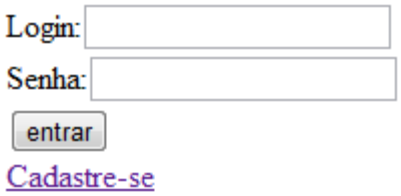

Figura 3. 7: Tela de login exibida para o usuário apartir do código anterior

Acima temos um exemplo de tela de login utilizando componentes JSF.

Ao clicar no botão "entrar" será disparado uma ação que será interceptada pelo controlador JSF que irá iniciar o ciclo de vida atualizando as informações e chamando o método de nome "logar" de uma classe java denominada "autenticador".

- **Conversão de dados:** As informações digitadas pelo usuário em formulários são em forma de texto. Porém, objetos de negócio exigem dados em forma de números, datas ou outros tipos. O JSF facilita a conversão de objetos e fornece mecanismos para conversão de objetos específicos.

- **Validação e manipulação de erros:** Mensagens de erros como "este campo é obrigatório" ou "este campo deve ser número" são facilmente implementadas com o seu mecanismo de validação de campos, além de fornecer formas de implementar mensagens personalizadas e novos validadores específicos.

- **Internacionalização:** O JSF gerência aspectos de internacionalização, como a codificação de caracteres(encondings) e a seleção de resource bundles(esquema de arquivo organizados em várias línguas do mundo).

- **Componentes customizados:** Os desenvolvedores têm condição de criar componentes sofisticados que podem ser simplesmente inseridos na página pelos web designers.

- **Suporte a AJAX:** O JSF oferece um canal de comunicação *Asynchronous Javascript and XML (*AJAX) padrão que executa de

maneira transparente a chamada das ações residentes no servidor sem a necessidade do desenvolvedor saber a fundo como funciona o AJAX.

3.4.6 Árvore de componentes

O controle que o JSF faz sobre o que acontece nos componentes só é possível porque o *framework* guarda e manipula os estados de cada componente em uma representação de objeto, organizando todos eles em uma árvore de componentes (SILVEIRA et al., 2012).

Cada elemento desta árvore guarda o estado atual dos componentes da tela e, a cada evento disparado, o controlador padrão JSF se encarrega de atualizar as informações nesta árvore.

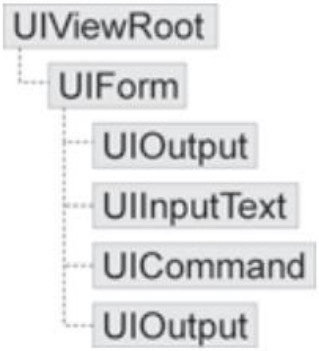

Figura 3. 8: Exemplo de um arvore de componentes JSF[25]

Com a utilização da árvore de componentes o JSF consegue mapear os dados da requisição do protocolo HTTP para todos os componentes da árvore, tirando o peso das costas do desenvolvedor de ficar tratando parâmetros de requisição. (SILVEIRA et al., 2012).

3.5 HIBERNATE

[25] http://videos.web-03.net/artigos/EvertonFoz/ConceitosAvancadosJSF/ConceitosAvancadosJSF3.jpg

Acessar bancos de dados relacionais numa perspectiva orientada a objetos é um requisito comum das aplicações atuais. Para realizar esta comunicação é necessário o mapeamento objeto-relacional (ORM), que visa preencher essa lacuna entre as estruturas de tabelas e visões e sua representação através de classes. Essa necessidade que veio a motivar a construção de *frameworks* de persistência de objetos, entre os mais populares está o *framework open-source* hibernate.[26]

3.5.1 Definição do Hibernate

O Hibernate é um *framework* ORM *open source* em Java, que abstrai o código SQL da aplicação, nos permitindo modificar a base de dados, sem grandes alterações no código Java.

O Hibernate é um framework de persistência que tem como finalidade armazenar objetos Java em bases de dados relacionais ou fornecer uma visão orientada a objetos de dados relacionais existentes (BAUER & KING, 2005).

"*É a tecnologia que provê a persistência de forma automatizada e transparente dos objetos em uma aplicação em tabelas de uma base de dados relacional, usando metadados para fazer a troca de dados entre os objetos e a base de dados.*"

(BAUER & KING, 2005).

3.5.2 Por que usar o Hibernate

Iniciado em 2001 por Gavin King, sua missão na época era simplesmente oferecer melhores capacidades de persistência simplificando as

[26] http://dsc.inf.furb.br/arquivos/tccs/monografias/2006-1odilonherculanosoaresfilhovf.pdf

complexidades e permitindo características faltantes no modelo tradicional oferecido na época.[27]

A Figura 3.9 fornece uma visão de altíssimo nível da arquitetura do Hibernate:

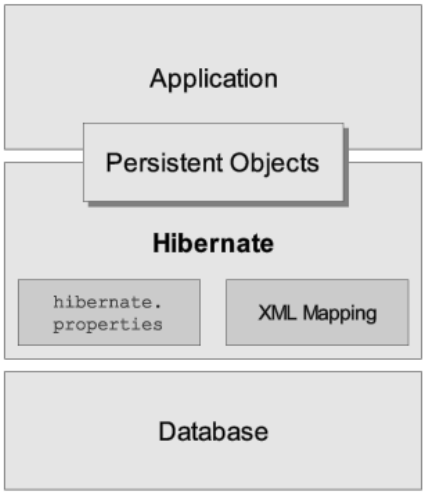

Figura 3. 9: Arquitetura do Hibernate[28]

Na figura 3.9 pode-se observar a aplicação com seus objetos persistentes se integrando com o Hibernate, configurado através do arquivo de propriedades e do arquivo XML. Observe que a aplicação não se comunica diretamente com o banco de dados, passando esta responsabilidade ao hibernate.

O Hibernate em diversos fatores tornou-se a solução para poupar tempo em muitas etapas de desenvolvimento por reduzir grandemente a quantidade de linhas de codificação beneficiando o tempo de desenvolvimento.

3.5.3 Java Persistence API

Até a versão 4, a plataforma java não possuía uma forma simples de mapear objetos em um banco de dados. Com o surgimento da especificação Java

[27] http://espweb.uem.br/site/files/tcc/2012/N%C3%A3o%20pagaram/Monografia%20EspWeb.pdf
[28] https://docs.jboss.org/hibernate/orm/3.5/reference/pt-BR/html/architecture.html

Persistence API (ou simplesmente JPA) foi possível padronizar o mapeamento de objeto-relacional. (GONÇALVES, 2007).

A JPA define um meio de mapeamento objeto-relacional para objetos java. Diversos *frameworks* ORM implementam a especificação JPA. O Hibernate nasceu sem JPA mas hoje em dia é comum acessar o Hibernate pela especificação JPA.

Como toda especificação, ela deve possuir implementações. Entre as implementações mais comuns, podemos citar: Hibernate da JBoss[29], EclipseLink da Eclipse Foundation[30] e o OpenJPA da Apache[31].

3.5.4 Mapeamento de Tabelas

O Hibernate oferece ao desenvolvedor a opção de criar mapeamentos entre os modelos de base de dados e modelos de objetos através de arquivos XML separados ou ainda através de anotações JPA no código fonte dos objetos.

Na Figura 3.10 podemos observar o mapeamento feito pelo Hibernate onde para cada tabela presente no banco de dados existe uma respectiva classe com seus atributos sendo representados pelas colunas da tabela. Cada registro (linha) da tabela é um objeto mapeado pelo Hibernate.

[29] http://hibernate.org/
[30] http://www.eclipse.org/eclipselink/
[31] http://openjpa.apache.org/

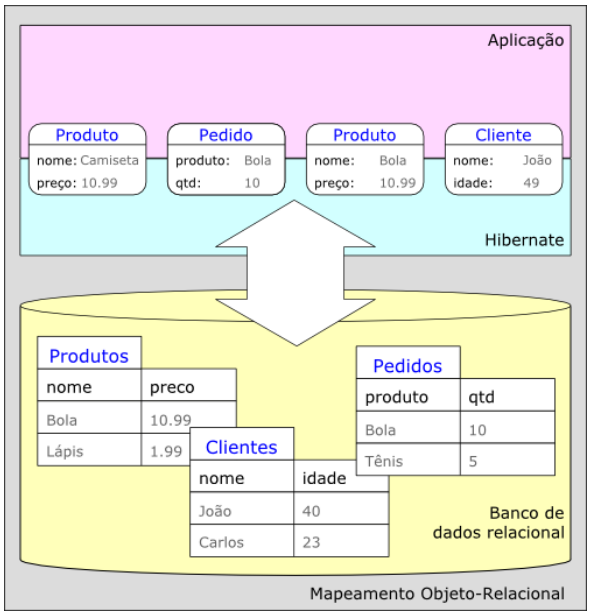

Figura 3. 10: Mapeamento objeto relacional[32]

Optar por arquivo XML externo ou annotations vai depender de cada caso, o uso de XML permite alterarmos o esquema da base de dados ou informações de mapeamento sem reconstruir a aplicação como um todo(sem ter que reiniciar o servidor). Com annotations temos um código mais intuitivo, com menos detalhes burocráticos em relação ao mesmo mapeamento no arquivo XML.

No projeto prático é feito uso de annotations pelo motivo de ser mais simples. Vale ressaltar também que grande parte das annotations utilizadas para o mapeamento não são de propriedade do Hibernate e sim da especificação Java Persistence API (JPA).

	id [PK] bigserial	celular character varying(16)	cpf_cnpj character varying(18)	dth_cadastro timestamp without time zone	dth_inativacao timestamp without time zone	email character varying(100)
1	1		999.999.999-99	2016-05-18 19:00:57.111		
2	2	(81) 9 9512-4241	082.046.974-22	2016-05-18 19:00:57.111		thiago@thiagoti.com
3	3	(81) 9 4567-3441	182.654.456-12	2016-05-18 19:00:57.111		raquel@thiagoti.com
4	4	(81) 9 5427-8748	486.5487.185-60	2016-05-18 19:00:57.111		marcos@thiagoti.com
5	5	(81) 9 9512-4241	234.547.122-88	2016-05-18 19:00:57.111		maria@teste.com
*						

Figura 3. 11: Exemplo de tabela tb_pessoa

[32] http://www.k19.com.br/artigos/wp-content/uploads/2010/08/orm.png

No fragmento de código abaixo temos um exemplo de como seria o mapeamento da classe Pessoa para a tabela da Figura 3.11 utilizando annotations JPA.

```
@Entity
public class Pessoa {
	@Id
	@GeneratedValue
	private Long id;

	@Column(length=100, nullable=false)
	private String nome;

	@Column(name="cpf_cnpj", length=18, nullable=false,
unique=true)
	private String cpfCnpj;

	@Column(length=16)
	private String celular;


	@Column(name="dth_cadastro", nullable=false)
	@Temporal(TemporalType.TIMESTAMP)
	private Date dataCadastro;

	//outros atributos...
}
```

A annotation @Entity indica que objetos dessa classe se tornem "persistível" no banco de dados. @Id indica que o atributo id é nossa chave primária, e @GeneratedValue diz que queremos que esta chave seja populada pelo banco (auto increment ou sequence, dependendo do banco de dados). Com @Temporal configuramos como mapear um atributo do tipo Date. Essas anotações precisam dos devidos imports, e pertencem ao pacote javax.persistence. Existem várias outras annotações, seu uso irá depender de cada caso.

3.5.5 O arquivo persistence.xml

O persistence.xml é um arquivo de configuração padrão da JPA. Nele deve ser definido as unidades de persistência, conhecidas como *persistence units.*

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
	xmlns="http://xmlns.jcp.org/xml/ns/persistence"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
      xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

      <persistence-unit name="com.thiagoti.erp">
            <provider>org.hibernate.ejb.HibernatePersistence</provider>

            <properties>
                  <property                    name="javax.persistence.jdbc.url"
value="jdbc:postgresql://localhost/erp" />
                  <property  name="javax.persistence.jdbc.user"  value="postgres"
/>
                  <property   name="javax.persistence.jdbc.password"   value="***"
/>
                  <property                 name="javax.persistence.jdbc.driver"
value="org.postgresql.Driver" />

                  <property name="hibernate.hbm2ddl.auto" value="update" />
                  <property name="hibernate.show_sql" value="true" />
                  <property                                name="hibernate.dialect"
value="org.hibernate.dialect.PostgreSQLDialect" />

                  <property           name="hibernate.connection.provider_class"
value="org.hibernate.connection.C3P0ConnectionProvider" />
                  <property name="hibernate.c3p0.max_size" value="20" />
                  <property name="hibernate.c3p0.min_size" value="5" />
                  <property name="hibernate.c3p0.acquire_increment" value="1" />
                  <property   name="hibernate.c3p0.idle_test_period"   value="300"
/>
                  <property name="hibernate.c3p0.max_statements" value="50" />
                  <property name="hibernate.c3p0.timeout" value="300" />
            </properties>
      </persistence-unit>

</persistence>
```

No XML acima tempos uma unidade de persistência do projeto prático deste trabalho, chamada *com.thiagoti.erp* com seus detalhes de configuração.

3.5.6 Exemplo de utilização do Hibernate com JPA

Para realizar acesso ao banco de dados utilizando o Hibernate com JPA é necessário trabalhar com uma *EntityManager*.

Entity Managers são configurados para serem capazes de persistir ou gerenciar tipos específicos de objetos, ler e escreve-los numa base de dados e ser implementado por um provedor de persistência, que no nosso caso trata-se do Hibernate.[33]

[33] http://www.devmedia.com.br/introducao-ao-entitymanager/5206

Todos *Entity Managers* vem de fábricas do tipo *EntityManagerFactory*. *EntityManagerFactory* que são definidas apartir do arquivo persistence.xml mostrado anteriormente.

```
EntityManagerFactory factory =
Persistence.createEntityManagerFactory("com.thiagoti.erp");

EntityManager entityManager = factory.createEntityManager();
```

No fragmento de código acima podemos observar a criação de uma fábrica de *EntityManager* e logo em seguida a criação de uma *EntityManager*.

De posse da *EntityManager* é possível interagir com o banco de dados.

```
Pessoa pessoa = new Pessoa();
pessoa.setNome(“João”);

entityManager.persist(pessoa);
```

O fragmento de código acima insere um novo registro no banco de dados, observe que nenhum SQL foi necessário ser implementado pelo desenvolvedor, tornando transparente a utilização.

3.6 INJEÇÃO DE DEPENDÊNCIA E CONTEXTO

Fowler (2004) apresenta em seu artigo *Inversion of Control Containers and the Dependency Injection pattern*[34] o termo injeção de dependência como uma técnica para prover dependências externas a um determinado componente de software.

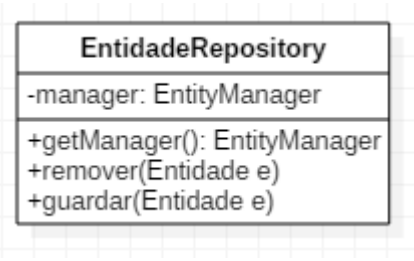

Figura 3. 12: Diagrama da classe EntidadeRepository

[34] http://martinfowler.com/articles/injection.html

Observando a Figura 3.12 podemos perceber que o objeto do tipo *EntidadeRepository* depende de um objeto *EntityManager* para seu funcionamento.

Com isso podemos afirmar que classe *EntidadeRepository* tem uma dependência externa, e em local do código que se tem algo parecido com o fragmento de código abaixo:

```
manager = createEntityManager();
```

A medida em o sistema cresce, torna-se difícil mantê-lo organizado é surge a necessidade de um auxílio. Precisamos de uma ferramenta que nos dê a possibilidade de montar nossos objetos a partir de outros, como um “lego”.

“Don’t call us, we’ll call you”, essa é a regra seguida pelo CDI. (PILGRIM, 2013).

3.6.1 Definição de Injeção de Dependência e Contexto

Injeção de Dependência e Contextos (CDI) é a especificação do java que cuida da parte de injeção de dependências. E, além de ser oficial e estar incluída em todos os servidores de aplicação java, é uma solução boa e produtiva.[35]

Esta solução faz com que as dependências entre os módulos(componentes de *software*) não sejam definidas programaticamente, mas sim pela configuração de uma infraestrutura de *software* (container) que é responsável por "injetar" em cada componente suas dependências declaradas.

3.6.2 Por que usar CDI

Abaixo é listado vantagens no uso desse mecanismo:

[35] http://blog.caelum.com.br/use-cdi-no-seu-proximo-projeto-java/

- **Simplicidade no código:** Com a injeção de dependência, os objetos da aplicação são liberados da tarefa de trazer suas próprias dependências (redução de código para inicialização “new” e configuração de dependências), e apenas focam na regra de negócio;
- **Baixo acoplamento entre os objetos:** O objeto conhece suas dependências apenas por sua interface pública (não por sua implementação, nem por como foram instanciadas), no futuro é possível realizar a troca sem maiores problemas;
- **Flexibilidade:** Com relação ao ciclo de vida dos objetos gerenciados pelo container CDI é possível realizar cache, o que melhora o desempenho do sistema, tornar o objeto *Singleton*(única instância do objeto para toda a aplicação), ou ter uma tempo de vida curta gerenciada automaticamente pelo container;

3.6.3 Escopos dos objetos gerênciados pelo CDI

Por padrão, toda dependência no CDI possui um escopo chamado *Dependant*. Isso significa que objetos serão instanciados tantas vezes quanto quem estiver chamando for instanciado.

Porém existem outros escopos comuns, como:

- ***RequestScoped*** que instância apenas um objeto que deverá ficar ativo durante toda a requisição feita pelo usuário;.
- ***SessionScoped*** que instância apenas um objeto que deverá ficar ativo durante toda a sessão do usuário;
- ***ApplicationScoped*** que instância apenas um objeto que deverá ficar ativo durante todo momento que a aplicação estiver rodando no servidor;

Escolher entre qual escopo utilizar irá depender do propósito do componente (objeto).

No fragmento de código abaixo é mostrado o exemplo de implementação da classe *EntidadeRepository* mencionada anteriormente.

```
	public class EntidadeRepository<E extends Entidade> implements
	Serializable {
		/**
		 *
		 */
		private static final long serialVersionUID = 1L;

		@Inject
		private EntityManager manager;

		public final void remover(E e){
			manager.remove(e);
		}
		public final void guardar(E e){
			E save = manager.merge(e);
			e.setId(save.getId());
		}
		protected EntityManager getManager() {
			return manager;
		}
}
```

Observe que em nenhum momento precisou ser necessário instanciar objeto do tipo *EntityManager*, abstraindo para o desenvolvedor a origem do objeto.

Através da annotation *@Inject* o CDI sabe que deve gerenciar o ciclo de vida do objeto, instanciando automaticamente quando necessário.

3.6.4 Produções de Objetos com CDI

Sempre que é encontrado um ponto de injeção *(@Inject)*, o container CDI tentará instanciar o objeto do tipo da classe que está sendo annotada.

Contudo, existem alguns objetos que necessitam de detalhes particulares para sua criação, como é o caso da classe *EntityManager*.

Em CDI, quando precisamos configurar a forma como o objeto deve ser instanciado utilizamos a annotation *@Produces* em um método que retorne o tipo do objeto solicitado.

No fragmento de código abaixo podemos observar como seria a implementação para instanciar um objeto do tipo *EntityManager*:

```
@ApplicationScoped
public class EntityManagerProducer {

	private EntityManagerFactory factory;

	public EntityManagerProducer() {
		factory 
Persistence.createEntityManagerFactory("com.thiagoti.erp");
	}

	@Produces @RequestScoped
	public EntityManager createEntityManager() {
		EntityManager manager = factory.createEntityManager();
		return manager;
	}

	public void closeEntityManager(@Disposes EntityManager manager) {
		manager.close();
	}

}
```

Temos uma classe produtora de *EntityManager singleton*(definido pela annotation *@ApplicationScoped*), ou seja, apenas uma por aplicação, onde cada *EntityManager* criado tem um escopo de requisição(definido pela annotation *@RequestScoped*), ao final da requisição, antes do CDI destruir o objeto será chamado o método *closeEntityManager* que irá liberar os recursos suportados pelo *EntityManager* (chamando o método *close*).

3.7 JASPER REPORT

Se analisarmos um ponto importante na maioria dos sistemas corporativos, sejam eles complexos ou simples, o objetivo final é quase sempre disponibilizar ao cliente uma grande quantidade de relatórios para que este possa então tomar a decisão correta.

Diversos sistemas geram algum tipo de relatório para seus clientes. Exemplos são listados:

- Um sistema de controle de bolsa de valores pode apresentar diversos gráficos e tabelas com dados importantes sobre as cotações atuais.
- Sistemas de prevenção de desastres naturais, onde capturam dados geográficos e geram relatórios para uma futura analise por parte do meteorologista.

- Um sistema de planejamento de recursos corporativo (ERP - *Enterprise Resource Planning*) que controla toda a gestão de uma empresa, esses sistemas normalmente oferecem diversos relatórios com base nos dados cadastrados.

Em java existe um *framework* utilizados para a geração de relatórios, o JasperReport[36].

3.7.1 Definição do Jasper Report

JasperReport é um poderoso e flexível gerador de relatórios *open source* que permite a geração de relatórios em diversos formatos como PDF, HTML, XLS, CSV e XML e com vários recursos.[37]

É possível criar relatórios utilizando a própria linha de comando java(com o auxílio das classes do JasperReport) ou utilizando arquivos XML com a extensão .jrxml.

3.7.2 Por que usar Jasper Report

O JasperReport com o auxílio da ferramenta gráfica iReport tem como objetivo acelerar o desenvolvimento de relatórios e oferecer diversos recursos como gráficos, tabelas, imagens, sumarização automática de valores, entre outros, que tornam o layout do relatório muito mais rico e de fácil manutenção.

3.7.3 IReport

A criação de um JasperReport manipulando XML ou programação direta acaba por se tornar uma tarefa complexa e não muito produtiva.

[36] http://community.jaspersoft.com/project/jasperreports-library
[37] http://www.devmedia.com.br/java-reporting-com-jasperreports-e-ireport-open-source-parte-iv/1745

Visando esta deficiência foi criado o iReport, que é uma ferramenta gráfica onde é possível desenharmos o relatório arrastando e soltando (*drag-and-drop*) diversos componentes sem nenhum conhecimento em XML. [38]

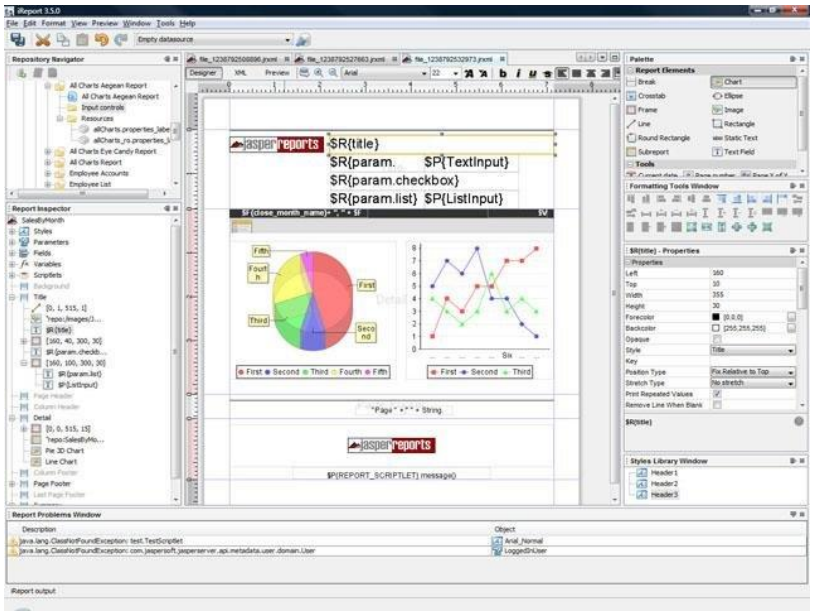

Figura 3. 13: Ferramenta IReport, utilizada para auxílio de arquivos JasperReport[39]

3.8 BOOTSTRAP

Nos dias atuais onde existe diferentes dispositivos com diferentes tamanhos de larguras de tela acessando a web nos mais diversos sites, é importante que sistemas ofereçam uma forma amigável de exibir suas informações.

3.8.1 Definição de Bootstrap

Em seu site oficial[40] na web, "*Originalmente criado por um designer e um desenvolvedor do Twitter, Bootstrap tornou-se um dos mais populares*

[38] http://www.devmedia.com.br/java-reporting-com-jasperreports-e-ireport-open-source-parte-i/1686
[39] https://a.fsdn.com/con/app/proj/ireport/screenshots/211684.jpg
[40] http://getbootstrap.com/about/

frameworks front-end e projetos de código aberto do mundo." (traduzido pelo autor).

É um conjunto de classes CSS e códigos JavaScript criados para facilitar o desenvolvimento de sites e sistemas web no lado cliente.

3.8.2 Por que usar o Bootstrap

Compatível com HTML5 e CSS3 (que são tecnologias atuais utilizadas no desenvolvimento web), o *framework* possibilita a criação de layouts responsivos(onde os elementos exibidos se ajustem automaticamente ao tamanho da largura da tela do dispositivo) e o uso de grids, permitindo que o conteúdo da página seja organizado em até 12 colunas e que comporte-se de maneira diferente para cada resolução de tela.[41]

3.8.3 Sistema de grids

O sistema de grids do Bootstrap possibilita a divisão em até 12 colunas de mesma largura em uma única linha, vejamos a Figura 3.14:

Figura 3. 14: Sistema de grids do Bootstrap[41]

[41] http://thiagonasc.com/desenvolvimento-web/desenvolvendo-com-bootstrap-3-um-framework-front-end-que-vale-a-pena

Caso deseje usar apenas uma coluna, basta defini-la com largura de tamanho doze(12) (que é a soma do total de colunas existentes).

A mesma lógica vale para exibição de duas colunas na tela, basta configurar duais colunas com largura de seis(6).

Abaixo tempos um fragmento de código HTML utilizando as classes CSS do bootstrap.

```
<div class="ui-grid-row">
      <div class="ui-grid-col-12">
      </div>
</div>
<div class="ui-grid-row">
      <div class="ui-grid-col-6">
      </div>
      <div class="ui-grid-col-6">
      </div>
</div>
<div class="ui-grid-row">
      <div class="ui-grid-col-4">
      </div>
      <div class="ui-grid-col-4">
      </div>
      <div class="ui-grid-col-4">
      </div>
</div>
```

A Figura 3.15 pode-se observar como seria a exibição do código HTML acima no navegador do usuário.

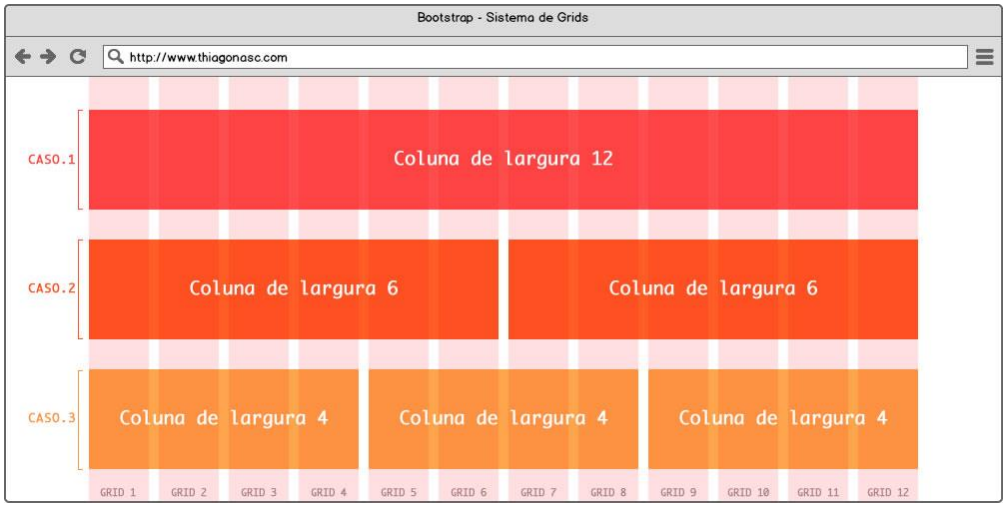

Figura 3. 15: Exemplo de utilização dos grids bootstrap[41]

Organizando o layout em grids o *framework* consegue adaptar a apresentação os componentes para diferentes larguras de tela.

Figura 3. 16: Elementos sendo exibidos em diferentes dispositivos[41]

4 ESTUDO DE CASO

Este capítulo descreve as atividades empregadas no processo de desenvolvimento do projeto. Destacando decisões de projeto importantes para o desenvolvimento e funcionamento do mesmo.

4.1 DOMÍNIO DO PROJETO

O projeto *Flat ERP* tem como objetivo gerenciar de todas as operações diárias de uma empresa. Trata-se de um ERP, sigla em inglês que significa *Enterprise Resource Planning*, que no português seria, Planejamento dos Recursos da Empresa.

A construção de um ERP é um processo que requer bastante tempo de desenvolvimento. Contudo, o estudo de caso realizado explora as técnicas e ferramentas discutidas no capítulo anteriores, este projeto compreende um protótipo com os módulos de cadastro de produtos, cadastro de clientes, cadastro de contas, cadastro de perfis de usuário, módulo de cadastro de pedido de venda e emissão de documento de venda além de uma tela para geração de relatórios de venda.

4.2 DESENVOLVIMENTO DA APLICAÇÃO

A aplicação foi desenvolvida utilizando o Windows 7 64bit, na IDE Eclipse Mars 1, e como servidor de aplicação foi utilizado o Tomcat 8.

4.2.1 Banco de Dados

O banco de dados escolhido para o sistema foi o PostgreSQL na versão 9.4, por ser extremamente robusto e confiável, além de ser extremamente rico em recursos, trata-se de um banco de dados relacional.[42]

Foi criado uma base de dados chamada *"erp"* e nela foram adicionadas tabelas que representam as entidades de negócio da aplicação, dentre elas pode-se destacar:

- **tb_pessoa:** Armazena todos os registros de pessoas FISICA ou JURIDICA, sendo assim, clientes, funcionários, fornecedores entre outros.
- **tb_produto:** Armazena todos os produtos comercializados.
- **tb_operacao_estoque:** Armazena todas as vendas registradas pelo sistema. Note que aqui, optou-se por um nome de tabela mais genérico, com o objetivo futuro de registrar também nesta tabela outros documentos, como Entrada de Produtos, Saída para Transferência, entre outros.
- **tb_operacao_estoque_produto:** Armazena os produtos da venda.
- **tb_lancamento:** Armazena todos os lançamentos financeiro gerados no sistema. Todo lançamento tem um tipo que pode ser Conta a Pagar ou Conta a Receber e trata-se de um agrupador de títulos(conta).
- **tb_titulo:** Armazena todos os títulos(parcelas de conta) gerados pelo sistema.
- **tb_baixa:** Armazena todas as baixas(pagamentos de parcelas) geradas pelo sistema.
- **tb_movimentacao_financeira:** Armazena toda manipulação financeira, entrada ou saída de dinheiro em alguma conta.

[42] https://www.ibm.com/developerworks/br/opensource/library/os-postgresecurity/

- **tb_conta:** Armazena todas as contas gerenciadas pelo sistema. Conta podem ser do tipo CAIXA ou BANCÁRIA.
- **tb_movimentacao_estoque:** Armazena toda manipulação de produtos, entrada ou saída de produtos do estoque da empresa.

Com o uso do Hibernate a criação da base de dados e das tabelas citadas se fez de forma automática, bastando configurar o arquivo persistence.xml com a seguinte opção:

```
<property name="hibernate.hbm2ddl.auto" value="update" />
```

Esse recurso ajudou na produtividade eliminando a tarefa de se criar tabelas utilizando o SQL direto no banco de dados.

Com o uso do Hibernate o desenvolvimento se fez focado apenas na plataforma java, sem se preocupar com SQL.

4.2.2 Entidades de Negócio

No projeto prático foi criado as classes de modelo (classes que representam as entidades de negócio do sistema), basicamente essas classes possuem como atributos as colunas de suas respectivas tabelas do banco de dados.

No diagrama de classes da Figura 4.1 pode-se observar as principais classes de negócio do sistema.

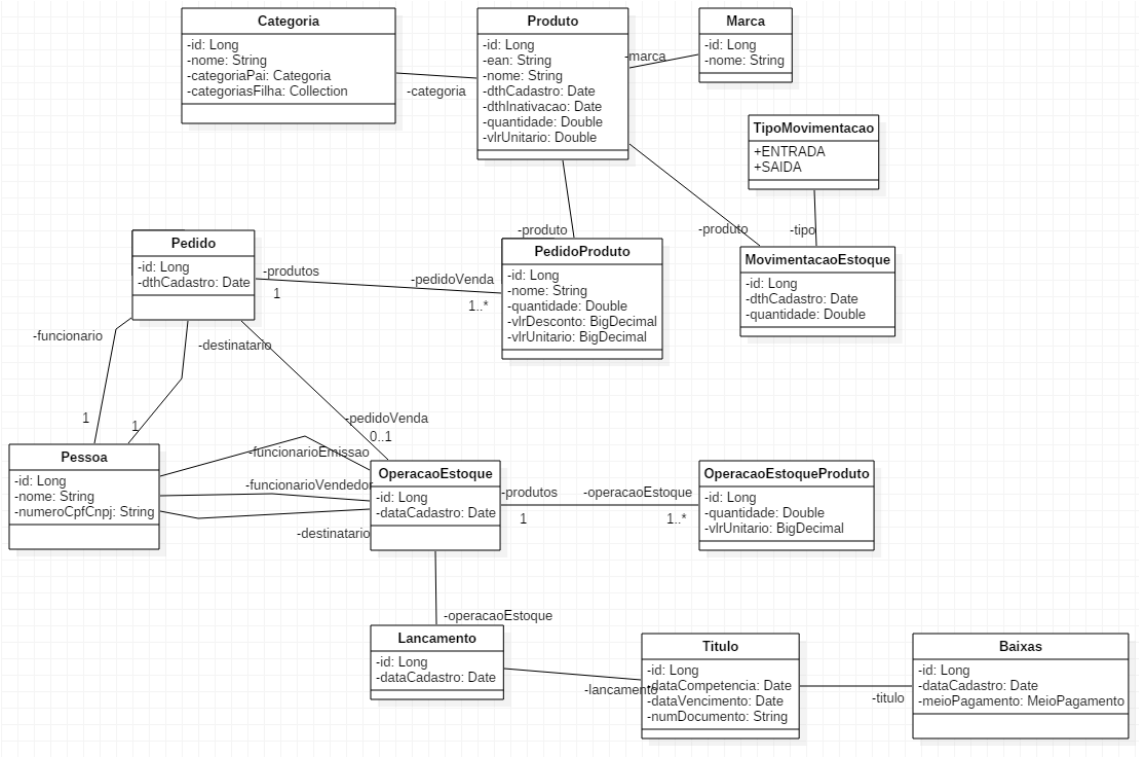

Figura 4. 1: Diagrama de classe do sistema Flat ERP

- **Pessoa:** representa uma pessoa FISICA ou JURÍDICA, um cliente, funcionário, fornecedor, entre outros.
- **Produto:** representa um produto que é comercializado pelo sistema.
- **OperacaoEstoque:** representa um documento de operação de estoque. Uma venda.
- **OperacaoEstoqueProduto:** representa um item presente no documento de operação de estoque.
- **Lancamento:** representa um lançamento financeiro em uma conta.
- **Titulo:** representa uma parcela de conta.
- **Baixa:** representa um pagamento de parcela.
- **MovimentacaoFinanceira:** representa uma movimentação de crédito ou débito em alguma conta.
- **Conta:** representa uma conta do tipo CAIXA ou BANCÁRIA onde são realizadas movimentações.

- **MovimentacaoEstoque:** representa uma movimentação de entrada ou saída no estoque de um produto.

4.2.3 Layout do Sistema

Para o desenvolvimento de um sistema web é necessário ter uma equipe especialista em *web design* para fornecer uma estrutura previamente definida de layout de sistema. Esta equipe normalmente é responsável por tratar de como o *layout* irá se comportar no navegador (Internet Explorer, Google Chrome entre outros), focando em tecnologias no lado cliente *(client-side).*

Uma outra solução, adotada por empresas de desenvolvimento de *software*, é conseguir o *layout* através de empresas terceirizadas, cobrindo apenas os custo para o desenvolvimento do mesmo.

Para o estudo de caso do presente trabalho, foi adotado um *layout* disponibilizado gratuitamente na web.[43]

O *layout* citado já oferece todo o suporte a tecnologias HTML, CSS e JavaScript, bastando apenas poucas adaptações que irá depender do objetivo de do projeto.

Na Figura 4.2 é apresentado a demonstração do *layout*.

[43] https://almsaeedstudio.com/AdminLTE

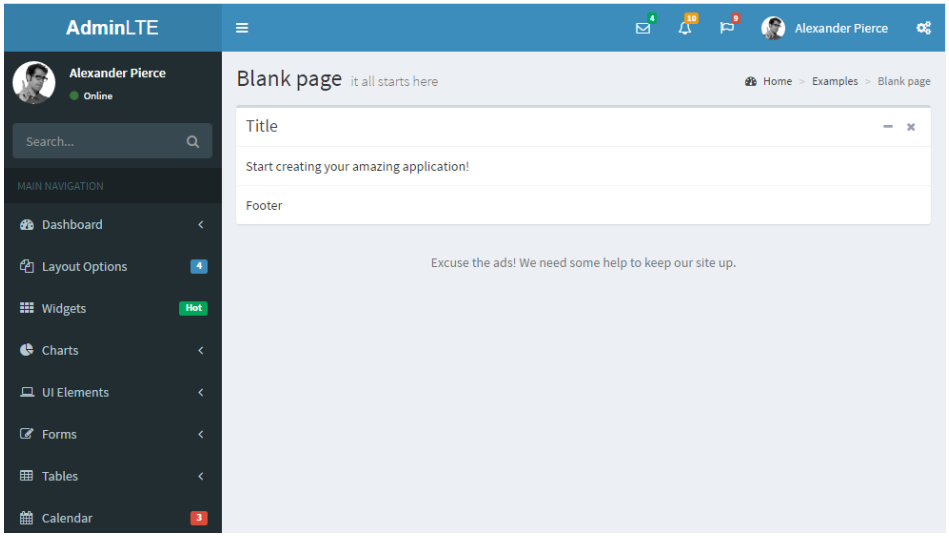

Figura 4. 2: Layout base do sistema, obtido de terceiro

4.2.4 Funcionalidades do Sistema

Tela de Login: Conforme representado na Figura 4.3 a tela de login é uma tela simples onde possibilita o usuário acessar o sistema informando seu login e senha de acesso.

Figura 4. 3: Tela de login do sistema

Tela Inicial: Conforme representado na Figura 4.4 a tela inicial do sistema é constituída pelo menu lateral esquerdo com módulos de cadastramento

e geração de relatórios. No centro é exibido uma linha do tempo com os eventos recentes. A direita é exibido um simples calendário marcando o dia atual.

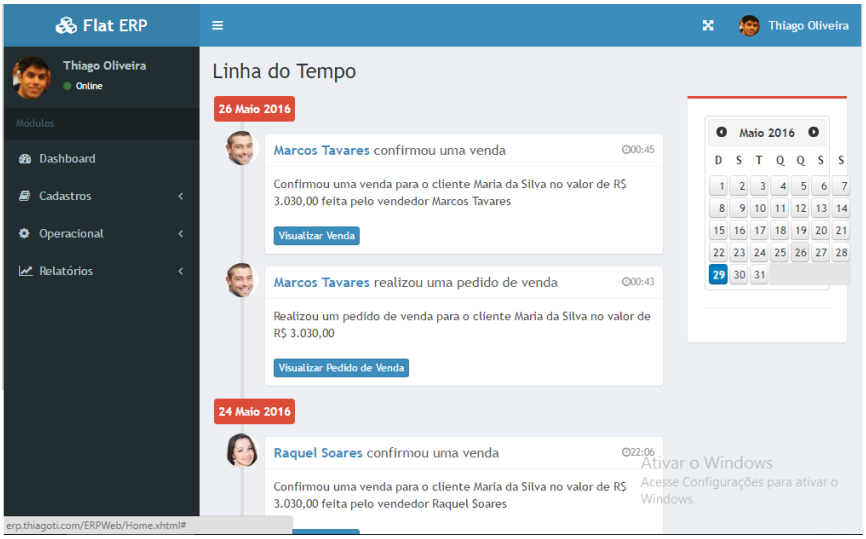

Figura 4. 4: Tela inicial do sistema

Tela Dashboard: Conforme representado na Figura 4.5 a tela de *dashbord* oferece vários gráficos reflexo dos dados cadastrados no sistema.

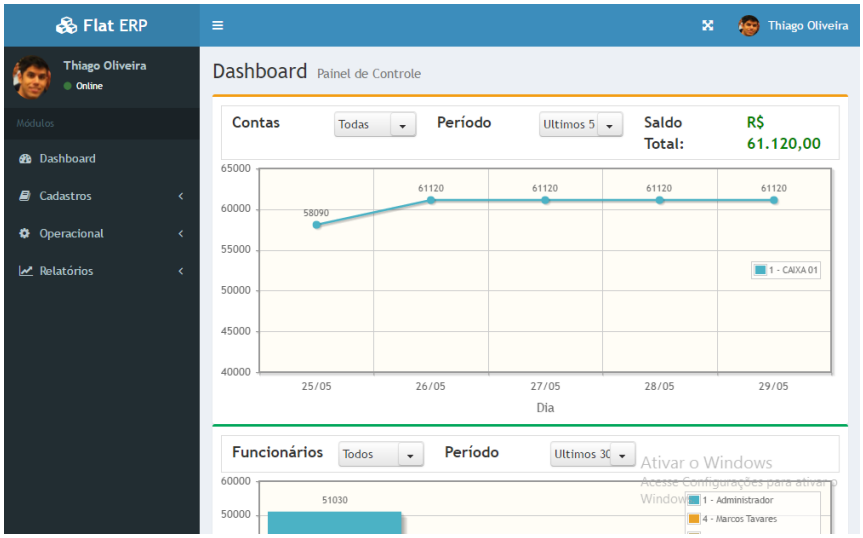

Figura 4. 5: Tela *Dashboard* do sistema

Tela de Cadastro de Produtos: Nesta tela é possível pesquisar, cadastrar ou alterar o cadastro de algum produto já existente.

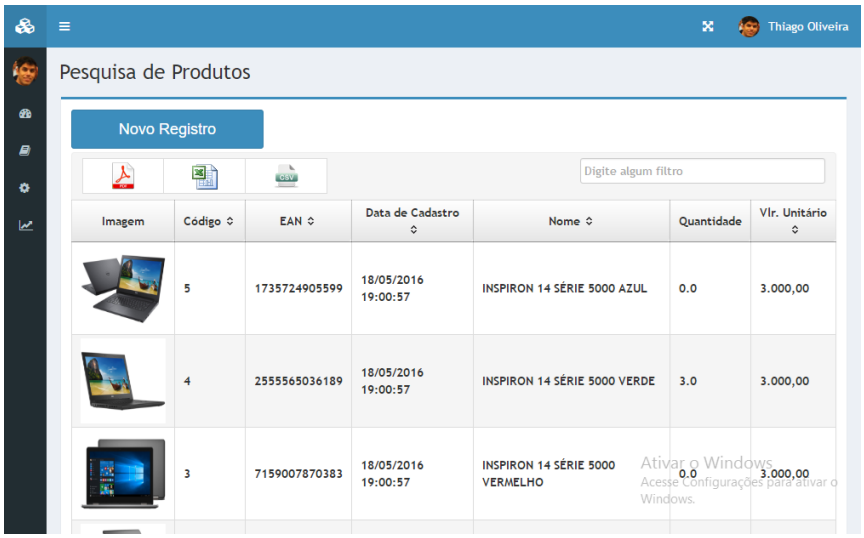

Figura 4. 6: Tela de Pesquisa de Produtos

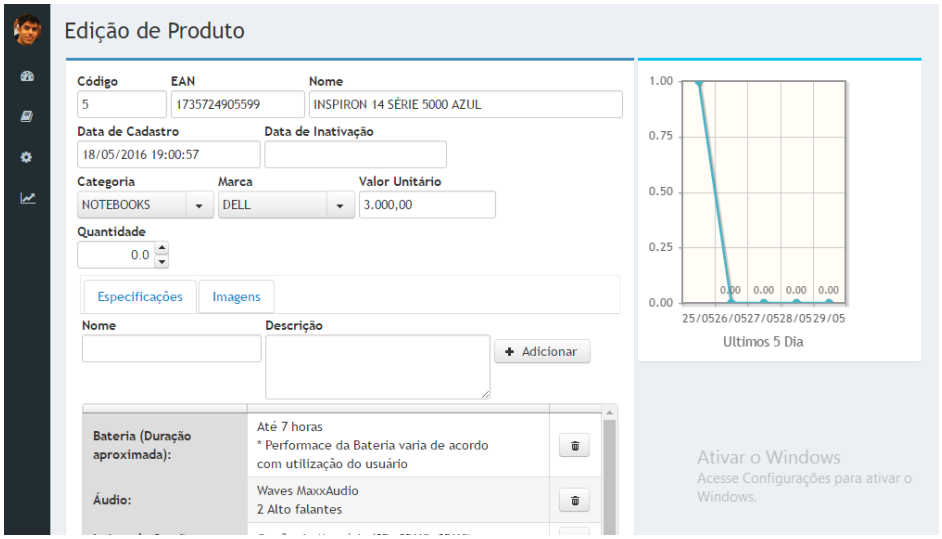

Figura 4. 7: Tela de Produtos – Manutenção de cadastro

Tela de Contas: Nesta tela é possível pesquisar, cadastrar ou alterar o cadastro de conta já existente.

Tela de Perfis de Usuário: Nesta tela é possível pesquisar, cadastrar ou alterar o cadastro de perfil já existente.

Tela de Pedido de Venda: Nesta tela é possível pesquisar, cadastrar ou alterar o cadastro de pedido de venda já existente.

Tela de Venda: Nesta tela é possível pesquisar ou cadastrar uma nova venda.

Tela de Relatório Vendas Por Período: Nesta tela é possível gerar um relatório das vendas cadastradas em um determinado período informado pelo usuário.

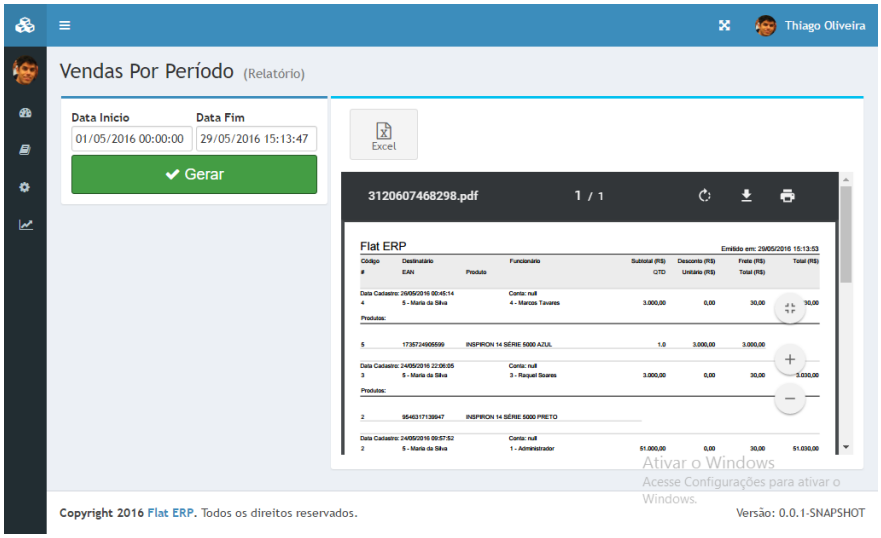

Figura 4. 8: Tela de relatório de Vendas por Período

4.2.5 Benefícios das ferramentas e técnicas estudadas

Abaixo é listado todas as ferramentas e técnicas estudas, aplicadas no projeto prático:

- **Arquitetura MVC:** Possibilitou a separação das responsabilidades entre a camada de apresentação e a lógica de negócio. Tornou-se uma arquitetura bastante flexível.
- **Bootstrap:** Utilizado no lado do cliente, na camada de apresentação, Layout do sistema independente de larga de tela de exibição. Design dos componentes(campos de texto, botões entre outros) moderno e pré-definido com uso de classes CSS padrões. Eliminação da necessidade do conhecimento profundo em JavaScript para realizar a lógica de ajuste em telas distintas.

- **Hibernate:** Utilizado no lado do servidor, na camada de acesso a dados. Possibilitou maior produtividade no desenvolvimento já que a comunicação com o banco de dados é de forma automática sem a preocupação do uso de SQL.

- **JSF:** Utilizado no lado do servidor, na camada de apresentação, Possibilitou maior produtividade no desenvolvimento com a abstração no tratamento de requisições HTTP; Uso de AJAX nativo e transparente para o desenvolvedor.

- **CDI:** Utilizado no lado do servidor, em todas as camadas da arquitetura. Possibilitou maior produtividade por oferecer uma gerência automática do ciclo de vida dos objetos. O desenvolvimento tornou-se livre da preocupação de instanciar objetos.

- **Primefaces:** Utilizado no lado do servidor, na camada de apresentação, Possibilitou maior produtividade no desenvolvimento com o uso de componentes(calendário, *dialogs* entre outros) ainda mais modernos e com recursos diferenciados. O desenvolvimento tornou-se livre da manipulação de JavaScript e CSS.

- **Padrões de Projeto:** Utilizado no lado do servidor, em todas as camadas da arquitetura. Possibilitou maior reuso da lógica de negócio. Entre os utilizados destaco o *Template Method* aplicado nas classes de controle de telas do JSF.

- **Codificação Limpa:** Utilizado no lado do servidor, em todas as camadas da arquitetura. Possibilitou maior organização e divisão de responsabilidade dos componentes. Seu uso abrange todo o código fonte do projeto.

5 CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho apresentou técnicas e práticas de Engenharia de Software para um desenvolvimento produtivo, flexível e de fácil manutenção.

No capítulo 2 explanou conceitos de arquitetura de *software* e sua importância no processo de desenvolvimento de um *software*, também apresentou pontos importantes do paradigma OO que podem fazer a diferença na construção de um código flexível. Ainda neste capítulo, em resumo, foi apresentado técnicas de codificação limpa que auxiliam em futuras manutenções e também conceitos de padrões de projeto de *softwares* orientado a objetos.

O capítulo 3 apresentou definições e vantagens na utilização de frameworks e também relatou com detalhes os frameworks utilizados no projeto prático deste trabalho.

Já o capítulo 4 apresentou o domínio de negócio do sistema web proposto como exemplo de implementação deste trabalho. Apresentou a modelagem dos componentes de negócio além das funcionalidades oferecidas pelo sistema web.

Diante das tecnologias utilizadas para realizar este trabalho foi possível chegar a uma conclusão que o uso de *framework* permite reduzir o processo repetitivo de desenvolvimento atribuindo maior tempo e atenção ao desempenho de tarefas que envolvem regras de negócios e métodos especiais.

O desenvolvimento no padrão arquitetural MVC (Modelo, Visão, Controle), facilitou a manutenção por separar a lógica de negócio da apresentação. Dando ao desenvolvedor um ambiente mais claro e objetivo de implementação no decorrer das atividades.

A utilização das bibliotecas de componentes Primefaces deu a aplicação uma apresentação visual rica e moderna, além de fornecer componentes úteis para a usabilidade do sistema, poupando a necessidade de um especialista em web designer.

O uso de padrões de projeto possibilitou a flexibilidade e reusabilidade dos módulos do sistema e as técnicas de codificação limpa auxiliaram no entendimento do código facilitando a implementação das funcionalidades apresentadas.

O fator tempo foi o obstáculo de maior influência no desenvolvimento no projeto prático.

O projeto prático, por se tratar de um protótipo, necessita de outros módulos para ser considerado uma versão passiva de implantação no cliente. Assim, o desenvolvimento

de módulos financeiros, e uma variedade maior de relatórios que auxilie o cliente a analisar os dados são funcionalidades que podem ser implementadas em um outro momento.

Também se pode pensar na criação de relatório dinamicamente, onde com a aplicação de técnicas de padrões de projeto torna-se possível oferecer para o usuário a possibilidade da exibição de relatórios dinâmicos, aumentando assim a possibilidade de apresentação dos dados ao cliente.

O desenvolvimento de uma camada *desktop* torna-se ainda mais fácil, já que toda a regra de negócio se encontra isolada(benefício da arquitetura MVC). Seria possível desenvolver uma camada *desktop* que realizasse vendas sem a necessidade do cliente utilizar o navegador web.

6 REFERÊNCIAS BIBLIOGRÁFICAS

ALEXANDER, CHRISTOPHER, **A Pattern Language**. Oxford USA Trade, 1977.

BARBOSA, 2001: **Middleware para Integração de Dados Heterogêneos Baseado em Composição de Frameworks**, tese de doutorado, Departamento de Informática PUC-Rio.

BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. 2. ed. **Software Architecture in Practice**. Addision-Wesley, 2003.

BAUER, Christian; KING, Gavin. **Hibernate in action**. Greenwitch: Manning Publications, 2005

CORDEIRO, Gilliard. **CDI Integre as dependências e contextos do seu código Java**. Casa do Código, São Paulo.

FAYAD, M. E. & JOHNSON, R. E.: **Domain-specific application frameworks: frameworks experience by industry**. New York: J. Wiley, 1999

FOWLER, Martin. **Padrões de Arquitetura de Aplicações Corporativas**. Bookman, 2006

FOWLER, Martin. **Analysis Patters – Reusable Object Models**. Addison-Wesley, 1997.

GAMMA, E., Helm, R., Johnson & R., Vlissides, J.: **Design patterns:**
elements of reusable object-oriented software. Addison-Wesley, Reading, MA, 1995.

GONÇALVES, E. **Desenvolvendo Aplicações Web com JSP Servlets, JavaServer Faces, Hibernate, EJB3 Persistence e Ajax**. Editora Ciência Moderna Ltda., 2007.

GUERRA, EDUARDO . **Design Patterns com Java - Projeto Orientado a Objetos Guiado por Padrões**. 1. ed. São Paulo: Casa do Códio, 2013.

KERIEVSKY, Joshua. **Refatoração para Padrões**. Porto Alegre: Bookman, 2008.

KRAFZIG, Dirk; BANKE, Karl; SLAMA, Dirk. **Enterprise SOA: Service-Oriented Architecture Best Practices**. Indianapolis: Prentice Hall, 2004.

MARTIN, Robert C. **Código Limpo: Habilidades Práticas do Ágile Software.** Rio de Janeiro-RJ: Alta Books, 2009.

MATTSSON, M.: **Object-oriented Frameworks - A survey of methodological issues"**, Licentiate Thesis, Department of Computer Science, Lund University, CODEN: LUTEDX/(TECS-3066)/1-130/(1996), also as Technical Report, LU- CS-TR: 96-167, Department of Computer Science, Lund University, 1996.

MELO, Maury. **Guia de Estudo para o Exame PMP: Project Management Professional Exam**. 4. ed. – Rio de Janeiro: Brasport, 2012.

PILGRIM, Peter A., **Java EE 7 Developer Handbook,** Packt Publishing, 2013.

PRESSMAN, R. **Engenharia de Software: Uma Abordagem Profissional**. 7ª Ed., São Paulo: McGraw-Hill, 2011.

SILVEIRA, Paul.et al., **Introdução à Arquitetura e Design de Software**, Casa do Codigo, 2012.

SOMMERVILLE, Ian. **Engenharia de Software.** – 9 ed. – São Paulo: Pearson Prentice Hall, 2011.