**DESEVOLVIMENTO DE UMA APLICAÇÃO WEB UTILIZANDO**

**FRAMEWORK SPRING MVC.**

Tobias P. A. Silva¹

Prof.: Edilson Lima2

**RESUMO**

Este artigo aborda o desenvolvimento de uma aplicação *Web* básica, em JAVA, adotando o *Framework* *Spring* MVC para gestão de negócios em uma empresa fictícia de aluguéis de veículos denominada LOCAUTO Locadora de Veículos. O sistema visa cadastrar clientes e veículos para locação e informatizar todo o processo de locação, que vai desde a reserva de um veículo até a sua devolução. O principal objetivo deste projeto é demonstrar de forma simples, como a utilização de um *Framework* pode acelerar o desenvolvimento de uma aplicação para *Web*, permitindo desenvolver rapidamente um sistema de software para melhorar as demandas operacionais desta empresa, facilitar o armazenamento e a recuperação de informações auxiliando de forma prática e funcional o cotidiano desse empreedimento. Este sistema foi concebido utilizando as ferramentas de programação para a linguagem JAVA, IDE Eclipse integrado ao Apache Maven e Servidor WAMP para persistência de dados no banco de Dados MySql.

**Palavras chaves:** JAVA, *Framework*, MySql.

1. **INTRODUÇÃO**

O objetivo precípuo deste sucinto artigo é demonstrar de forma clara e objetiva a aplicação de um *Framework* de padronização no desenvolvimento de uma aplicação Web, que visa atender às demandas operacionais básicas de uma pequena empresa que trabalha no ramo de locações de veículos, automatizando tarefas rotineiras e oferecendo mais agilidade nas tomadas de decisões.

* 1. **Definição do problema**

Com um significativo aumento no número de locações de veículos no país, surgiram para as empresas do ramo, vários problemas relacionados com as principais rotinas operacionais que envolvem uma locação de um veículo, tais como:

* Cadastrar novos clientes;
* Cadastrar novos veículos;
* Gerenciar locações e devoluções, e;
* Gerar relatórios.

Uma vez que se tornou, muito custoso e demorado o processo de gerenciamento de todos estes dados. Outro importante problema reside no fato de que, atualmente, com o avanço tecnológico, era necessário oferecer aos clientes uma forma mais rápida e cômoda de efetuar a reserva de um veículo sem sair de casa utilizando um dispositivo conectado à Internet.

* 1. **Soluções e objetivos**

Após uma criteriosa análise dos problemas expostos anteriormente é possível determinar inúmeras soluções possíveis dentre as quais adotaremos a seguinte:

Criação de um sistema *Web* conectado a um banco de dados para gravação de dados, que permita cadastro de clientes e veículos disponíveis para locação, locações em andamento ou com atraso de devolução, calculo automatizado de totais e taxas relacionadas ao valor da diária praticada pela prestadora de serviço de locação.

Como mencionado anteriormente que o objetivo principal deste artigo era demonstrar a utilização do uso de um Framework no desenvolvimento de uma aplicação Web, as funcionalidades deste sistema serão mitigadas visando um enfoque mais preciso na parte prática na modelagem MVC utilizada pelo já citado *Framework*.

**1.3. Organização do trabalho**

Inicialmente, foram estabelecidas as principais funcionalidades que o sistema deveria possuir, em seguida foram elaborados os diagramas necessários para direcionamento correto para abstração dos requisitos funcionais e não funcionais em linhas de código de linguagem que oferecesse suporte a aplicações Web com persistência de dados. A fase final contempla testes finais e apresentação do produto final e soluções encontradas.

**2. FUNDAMENTAÇÃO TEÓRICA**

**2.1. PROCESSO UNIFICADO**

De acordo com Martins Vidal (1999) o processo unificado de desenvolvimento de software é o conjunto de atividades necessárias para transformar requisitos do usuário em um sistema de software. Ele é baseado em componentes, o que significa o sistema ser construído a partir de componentes de software interconectados via interfaces muito bem definidas. O processo unificado utiliza a Linguagem de Modelagem Unificada (*Unified* *Modeling Language* – UML) no preparo de todos os artefatos do sistema.

O Processo Unificado para o desenvolvimento dessa aplicação Web, iniciou-se pelo levantamento de requisitos e funcionalidades necessárias e desejáveis que o aplicativo deveria possuir, tais requisitos são obtidos por meio de pesquisas e em reuniões com os interessados diretos e prováveis usuários do sistema/aplicativo embrião. Mas neste caso, por se tratar de uma aplicação apenas para fins didáticos, iremos suprimir essa etapa. Na fase seguinte tem início a construção dos diagramas de classe, de casos de uso e identificação de papeis (*Roles*), dentre outros.

Depois de finda a fase de diagramação, pode-se proceder à parte de programação propriamente dita, que terá sempre como diretriz principal, os requisitos estabelecidos na primeira fase (levantamento de requisitos e funcionalidades) e os diagramas obtidos a partir destes.

Para esta fase final, utilizaremos como auxílio um software de codificação com suporte a linguagem JAVA, uma IDE (*Integrated Development Enviroment*), desenvolvida especificamente para este fim. Dentre as muitas disponíveis para uso, foi escolhida a IDE Eclipse, na sua versão mais recente *Oxigen,* a qual oferece suporte ao *Framework Spring*, possibilitando o uso do modelo de arquitetura de aplicação MVC. Para gerenciamento de dependências foi escolhido o APACHE MAVEN, justamente por sua facilidade de configuração e uso. E por fim para gravação e recuperação dos dados a serem processados, o sistema julgado adequado foi o WAMP *Server*, também por se tratar de uma solução gratuita, com excelente desempenho e facilidade de configuração e utilização.

**2.1.1. Características Básicas**

Como já foi dito anteriormente, por se tratar de uma aplicação simples, com fins apenas de aprendizagem e exemplificação, como características básicas desta aplicação, ficou estabelecido que a mesma deveria facilitar o cadastro de novos clientes, com possibilidade de posterior localização, alteração e, caso oportuno, exclusão do mesmo do banco de dados.

Além desse cadastro de clientes, de forma parecida, deve ser possível cadastro de veículos, com funcionalidades semelhantes ao cadastro de clientes, porém com características próprias inerentes a veículos tais como: ano, modelo, quilometragem, dentre outras.

Como funcionalidade principal, o sistema deve ser capaz de atrelar o cadastro de um cliente a um ou mais veículos, caracterizando a relação de locação, calculando de forma praticamente automática, com base em parâmetros previamente estabelecidos, valores totais de diárias e quando aplicáves, multas por atrasos e/ou quilometragem excedida além do limite estabelecido em contrato, caso ocorra.

Dentre as funcionalidades desejáveis estão a geração de relatórios de locações e devoluções próximas a ocorrer, bem como total de faturamento para um determinado período.

**2.2. CONCEITOS BÁSICOS DE POO**

O desenvolvimento de software é extremamente amplo. Nesse mercado, existem diversas linguagens de programação, que seguem diferentes paradigmas. Um desses paradigmas é a Orientação a Objetos, que atualmente é o mais difundido entre todos. Isso acontece porque se trata de um padrão que tem evoluído muito, principalmente em questões voltadas para segurança e reaproveitamento de código, o que é muito importante no desenvolvimento de qualquer aplicação moderna. Esse padrão se baseia em quatro pilares que veremos ao longo dos tópicos seguintes. Além disso, a POO, como costuma ser também denominada, possui diversas vantagens em sua utilização, que também serão vistas e explicadas.

**2.2.1. Objeto**

Segundo Felipe Eduardo (2016), objeto é a representação computacional de objetos do mundo real e são definidos como um conjunto de atributos (variáveis) e comportamentos (métodos). Em um programa orientado a objetos, podemos ter muitos objetos que se relacionam entre si. Essa interação entre eles é feita através de mensagens. Um objeto chama os métodos de outro, passando parâmetros quando necessário. Em um exemplo didático, para exemplificação podemos dizer que o objeto do tipo Pessoa passa a mensagem “mude a marcha” para um objeto do tipo Carro, passando como parâmetro a o número da marcha desejada.

**2.2.2. Classes de Objetos**

Segundo preceitua Ivan Ricarte (2000), Uma classe é um gabarito para a definição de objetos. Através da definição de uma classe, descreve-se que propriedades -- ou atributos -- o objeto terá. Além da especificação de atributos, a definição de uma classe descreve também qual o comportamento de objetos da classe, ou seja, que funcionalidades podem ser aplicadas a objetos da classe. Essas funcionalidades são descritas através de métodos.

Um ou mais objetos que tem características semelhantes pertencem a uma classe. Em programação orientada a objetos, antes de tudo construímos as classes e posteriormente as usamos para criar (instanciar) objetos daquele tipo.

Quando instanciamos um objeto, alocamos espaço na memória para podermos utilizar os atributos e métodos desse objeto, e fazemos isso através da palavra reservada ***new.***

Exemplo:

Cachorro cachorro1 = ***new*** Cachorro( );

Cachorro cachorro2 = ***new*** Cachorro( );

Desta forma estamos instanciando dois objetos do tipo Cachorro (Classe), que embora pertença, a mesma classe e possuírem características semelhantes cada um possui seu próprio conjunto de atributos.

**2.2.3. Abstração**

Segundo Henrique Machado (2015). A abstração consiste em um dos pontos mais importantes dentro de qualquer linguagem Orientada a Objetos. Como estamos lidando com uma representação de um objeto real (o que dá nome ao paradigma), temos que imaginar o que esse objeto irá realizar dentro de nosso sistema. São três pontos que devem ser levados em consideração nessa abstração.

O primeiro ponto é darmos uma identidade ao objeto que iremos criar. Essa identidade deve ser única dentro do sistema para que não haja conflito. Na maior parte das linguagens, há o conceito de pacotes. Nessas linguagens, a identidade do objeto não pode ser repetida dentro do pacote, e não necessariamente no sistema inteiro.

A segunda parte diz respeito a características do objeto. Como sabemos, no mundo real qualquer objeto possui elementos que o definem. Dentro da programação orientada a objetos, essas características são nomeadas propriedades. Por exemplo, as propriedades de um objeto “Cachorro” poderiam ser “Tamanho”, “Raça” e “Idade”.

Por fim, a terceira parte é definirmos as ações que o objeto irá executar. Esses métodos podem ser extremamente variáveis, desde “Acender ( )” em um objeto lâmpada até “Latir ( )” em um objeto cachorro.

**2.2.3.1. Interfaces**

De acordo com Willian Xavier (2010), interface é um recurso da orientação a objeto utilizado em Java que define ações que devem ser obrigatoriamente executadas, mas que cada classe pode executar de forma diferente. Interfaces contém valores constantes ou assinaturas de métodos que devem ser implementados dentro de uma classe.

E por que isso?

Isso se deve ao fato que muitos objetos (classes) podem possuir a mesma ação (método), porém, podem executá-la de maneira diferente. Usando um exemplo bem drástico, podemos ter uma interface chamada aéreo que possui a assinatura do método voar ( ). Ou seja, toda classe que implementar aéreo deve dizer como voar ( ).

Portanto, se eu tenho uma classe chamada pássaro e outra chamada avião, ambas implementando a interface aéreo, então, nestas duas classes devemos codificar a forma como cada um irá voar ( ). Uma interface é criada da mesma forma que uma classe, mas utilizando a palavra-chave *interface* no lugar de *class*.

Exemplo de nomenclatura de uma interface:

nomeDaInterface {

metodoAbstrato (argumentos);

}

**2.2.5. Encapsulamento**

Ainda segundo Henrique Machado (2015). O encapsulamento é uma das principais técnicas que define a programação orientada a objetos. Trata-se de um dos elementos que adicionam segurança à aplicação em uma programação orientada a objetos pelo fato de esconder as propriedades, criando uma espécie de caixa preta.

A maior parte das linguagens orientadas a objetos implementam o encapsulamento baseado em propriedades privadas, ligadas a métodos especiais chamados *getters* e *setters*, que irão retornar e determinar o valor da propriedade, respectivamente. Essa atitude evita o acesso direto a propriedade do objeto, adicionando a uma outra camada de segurança à aplicação.

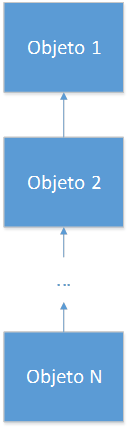
Para fazermos um paralelo com o que vemos no mundo real, temos o encapsulamento em outros elementos. Por exemplo, quando clicamos no botão ligar da televisão, não sabemos o que está acontecendo internamente. Podemos então dizer que os métodos que ligam a televisão estão encapsulados.

**2.2.6. Herança**

O reuso de código é uma das grandes vantagens da programação orientada a objetos. Muito disso se dá por uma questão que é conhecida como herança conforme pontua Henrique Machado (2015). Essa característica otimiza a produção da aplicação em tempo e linhas de código.

Para entendermos essa característica, vamos imaginar uma família: a criança, por exemplo, está herdando características de seus pais. Os pais, por sua vez, herdam algo dos avós, o que faz com que a criança também o faça, e assim sucessivamente. Na orientação a objetos, a questão é exatamente assim, como mostra a Figura 1. O objeto abaixo na hierarquia irá herdar características de todos os objetos acima dele, seus “ancestrais”. A herança a partir das características do objeto mais acima é considerada herança direta, enquanto as demais são consideradas heranças indiretas. Por exemplo, na família, a criança herda diretamente do pai e indiretamente do avô e do bisavô.

**Figura 1: Representação de Herança.**



Fonte:https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264.

**2.2.7. Polimorfismo**

Outro ponto essencial na programação orientada a objetos é o chamado polimorfismo. Na natureza, vemos animais que são capazes de alterar sua forma conforme a necessidade, e é dessa ideia que vem o polimorfismo na orientação a objetos. Como sabemos, os objetos filhos herdam as características e ações de seus “ancestrais”. Entretanto, em alguns casos, é necessário que as ações para um mesmo método seja diferente. Em outras palavras, o polimorfismo consiste na alteração do funcionamento interno de um método herdado de um objeto pai (Henrique Machado, 2015).

Como um exemplo, temos um objeto genérico “Eletrodoméstico”. Esse objeto possui um método, ou ação, “Ligar ( )”. Temos dois objetos, “Televisão” e “Geladeira”, que não irão ser ligados da mesma forma. Assim, precisamos, para cada uma das classes filhas, reescrever o método “Ligar ( )”.

Com relação ao polimorfismo, valem algumas observações. Como se trata de um assunto que está intimamente conectado à herança, entender os dois juntamente é uma boa ideia. Outro ponto é o fato de que as linguagens de programação implementam o polimorfismo de maneiras diferentes. O C#, por exemplo, faz uso de método virtuais (com a palavra-chave *virtual*) que podem ser reimplementados (com a palavra-chave *override*) nas classes filhas. Já em Java, apenas o atributo “@*Override*” é necessário.

Esses quatro pilares são essenciais no entendimento de qualquer linguagem orientada a objetos e da orientação a objetos como um todo. Cada linguagem irá implementar esses pilares de uma forma, mas essencialmente é a mesma coisa. Apenas a questão da herança, como comentado, que pode trazer variações mais bruscas, como a presença de herança múltipla. Além disso, o encapsulamento também é feito de maneiras distintas nas diversas linguagens, embora os *getters* e *setters* sejam praticamente onipresentes.

**2.3. LINGUAGEM DE MODELAGEM UNIFICADA (UML)**

A UML (*Unified Modeling Language*), ou Linguaguem de Modelagem Unificada em tradução livre, é um modelo de linguagem para modelagem de dados orientados a objetos muito utilizada para especificar, construir, visualizar, e documentar um sistema de software. Através dela é possível relazar uma modelagem visual de maneira que os relacionamentos entre os componentes do sistema sejam melhor visualizados e compreendidos. Tecnicamente falando, a UML é a junção das três mais conceituadas linguagem de modelagem orientados a objetos (Booch de Grady, OOSE de Jacobson e o OMT de Rumbaugh) (Admilson Nogueira,2017).

**2.3.1. Diagrama de Casos de Uso**

O diagrama de casos de uso é um diagrama da UML cujo objetivo é representar um requisito do sistema que será automatizado. Considere como requisito uma necessidade do sistema (Marcelo Madureira,2007). Usamos atores (figura 2) para representar as entidades que interagem com o sistema. Podem ser usuários, máquinas, sensores, etc… Um ator representa um papel no sistema, mas um papel pode ser representando por vários atores.

**Figura 2: Representação de um Ator.**



Fonte: Autoral.

**2.3.2. Diagrama de Classes**

O diagrama de classes é considerado por muitos autores como o mais importante e o mais utilizado diagrama da UML. Seu principal enfoque está em permitir a visualização das classes que irão compor o sistema com seus respectivos atributos e métodos, bem como em demonstrar como as classes do sistema se relacionam, se complementam e transmitem informações entre si. Este diagrama apresenta uma visão estática de como as classes estão organizadas, preocupando-se em definir a estrutura lógica das mesmas. O diagrama de classes serve como base para a construção da maior parte dos demais diagramas da UML (Paulo César, 2017).

**2.3.3. Diagrama de Sequência**

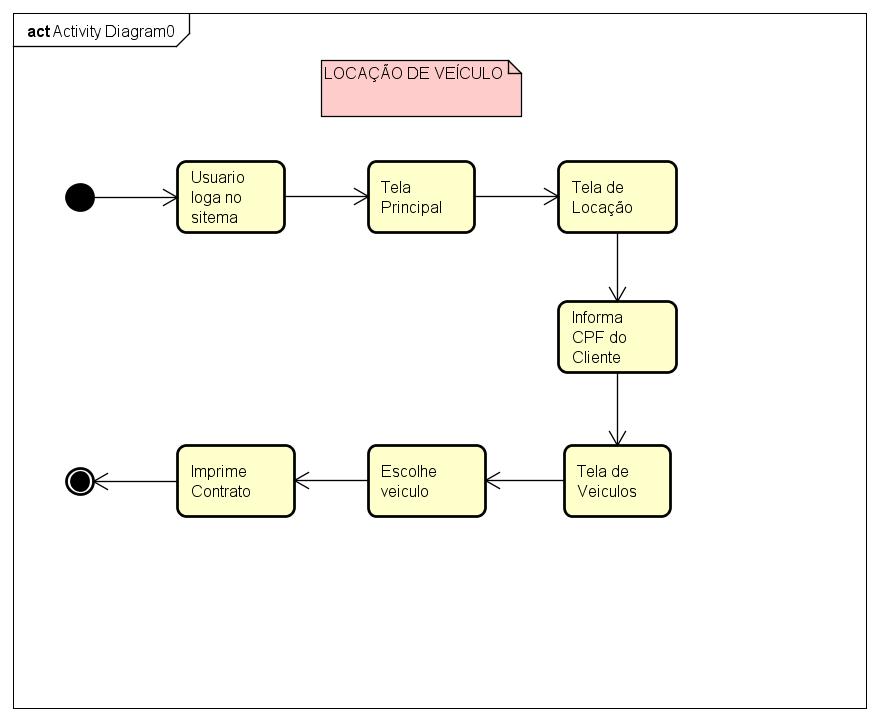
Um diagrama de sequência é um diagrama UML que ilustra a sequência das mensagens entre objetos em uma interação. Um diagrama de sequência consiste em um grupo de objetos representados por linhas de vida e as mensagens que eles trocam durante a interação (IBM, 2018). Um diagrama de seqüência mostra a sequência de mensagens transmitidas entre objetos. Diagramas de sequência também mostram as estruturas de controle entre objetos.

**2.3.4. Diagrama de Atividade**

No contexto da UML, o Diagrama de Atividades é um diagrama comportamental (que especifica o comportamento do software), e através dele podemos modelar partes do comportamento de um software. O diagrama de atividades, como citado, tem como objetivo principal a especificação do comportamento do software, do ponto de vista funcional, ou seja, das suas funcionalidades. É muito semelhante a um fluxograma, uma ferramenta utilizada há muitas décadas, principalmente na administração (Plínio Ventura, 2016).

Seguindo o tema do projeto abordado, a figura abaixo nos traz uma idéia geral de como se comportaria nosso sistema de locação para uma locação de um veículo.

**Figura 3: Diagrama de Atividade.**

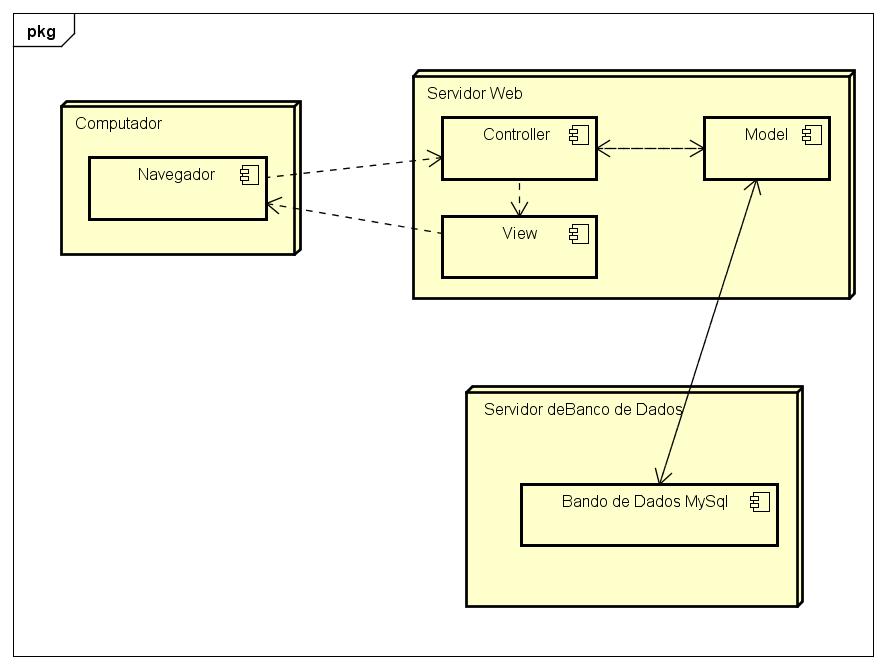
****

Fonte: Autoral.

**2.3.5. Diagrama de Implantação**

Na UML, os diagramas de implementação modelam a arquitetura física de um sistema. Os diagramas de implementação mostram os relacionamentos entre os componentes de software e hardware no sistema e a distribuição física do processamento. Os diagramas de implementação, que normalmente são preparados durante a fase de desenvolvimento da implementação, mostram a organização física dos nós em um sistema distribuído, os artefatos que estão armazenados em cada nó e os componentes e outros elementos que os artefatos implementam. Os nós representam dispositivos de hardware como por exemplo servidores, sensores e impressoras, bem como outros dispositivos que suportam o ambiente de tempo de execução de um sistema. Caminhos de comunicação e relacionamentos de implementação modelam as conexões do sistema (IBM, 2018).

Abaixo (figura 4) temos uma representação sucinta da infraestrutura necessária para implementar este sistema e deixá-lo apto para uso.

**Figura 4: Diagrama de Implantação.**

Fonte: Autoral.

**2.3.6. Ferramentas *CASE* Baseadas na linguagem UML**

São diversas as ferramentas disponíveis no mercado para a geração de diagramas UML. Existem desde soluções gratuitas e que contam com um bom suporte para a elaboração de representações baseadas nesta linguagem, passando ainda por softwares proprietários dotados de uma ampla gama de recursos (Renato José,2016).

No que se refere a aplicativos pagos, é comum que muitos destes ofereçam funcionalidades baseadas em mecanismos de engenharia reversa (para a geração de diagramas a partir de implementações pré-existentes) ou, até mesmo, a obtenção de código-fonte tomando por base diagramas concebidos a partir da ferramenta em questão (o que não significa que todo o esforço de codificação será substituído por esta última).

Constituem bons exemplos de aplicações que suportam a construção de diagramas baseados nas notações da UML:

* Astah UML: conhecida anteriormente como JUDE, esta solução conta tanto com versões gratuitas quanto pagas. É fornecida pela empresa japonesa Change Vision, disponibilizando recursos para a elaboração dos diferentes diagramas previstos pela UML;
* Enterprise UML: software de modelagem comercializado pela Sparx Systems, contando com total suporte à construção dos diferentes diagramas de UML, além de compatibilidade com diversas linguagens como Java e C# (geração de código e aplicação de engenharia reversa);
* Visio: parte integrante do pacote Office da Microsoft, este aplicativo também permite que diagramas UML sejam elaborados a partir do mesmo, além de um amplo conjunto de outros tipos de representações gráficas suportadas.

Importante destacar ainda que todas as ferramentas mencionadas nesta seção possibilitam a exportação de diagramas para diferentes formatos gráficos, tais como .gif, .jpg e .png. Todos os diagramas presentes neste artigos foram criados utilizando Astah UML.

**2.4. MVC**

MVC é nada mais que um padrão de arquitetura de software, separando sua aplicação em 3 camadas. A camada de interação do usuário(view), a camada de manipulação dos dados(model) e a camada de controle(controller) (Allan Ramos, 2015).

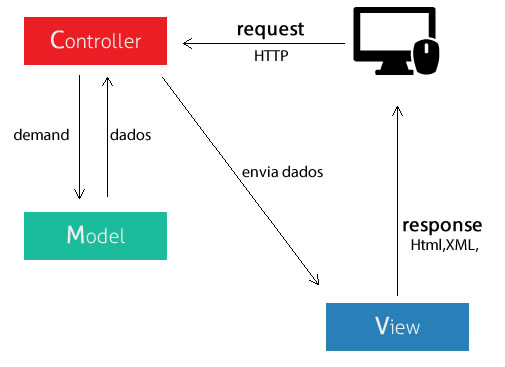
Model: Sempre que você pensar em manipulação de dados, pense em model. Ele é responsável pela leitura e escrita de dados, e também de suas validações.

View: Simples: a camada de interação com o usuário. Ela apenas faz a exibição dos dados, sendo ela por meio de um html ou xml.

Controller: O responsável por receber todas as requisições do usuário. Seus métodos chamados actions são responsáveis por uma página, controlando qual model usar e qual view será mostrado ao usuário.

Abaixo temos uma representação ilustrativa da arquitetura MVC.

**Figura 5: Arquitetura MVC.**



Fonte: https://tableless.com.br/mvc-afinal-e-o-que/.

**3. CASOS DE USO**

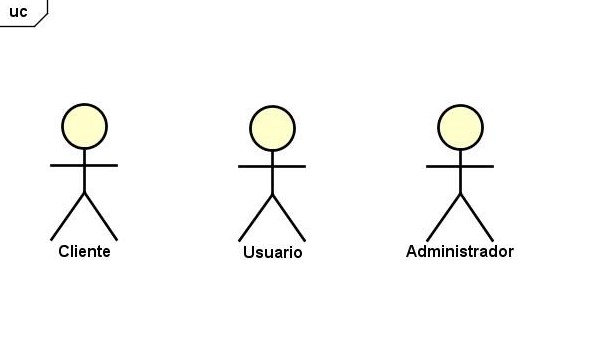
**3.1. Cenário da Aplicação**

**3.2. Análise da aplicação**

**3.2.1. Identificação de Atores**

Para o caso de nosso aplicativo *Web,* inicialmente foram identificados 3 atores principais, Cliente , Usuário e Administrador, cada um deles com suas propias ações específicas que seram detalhadas mais adiante no diagrama de casos de uso.

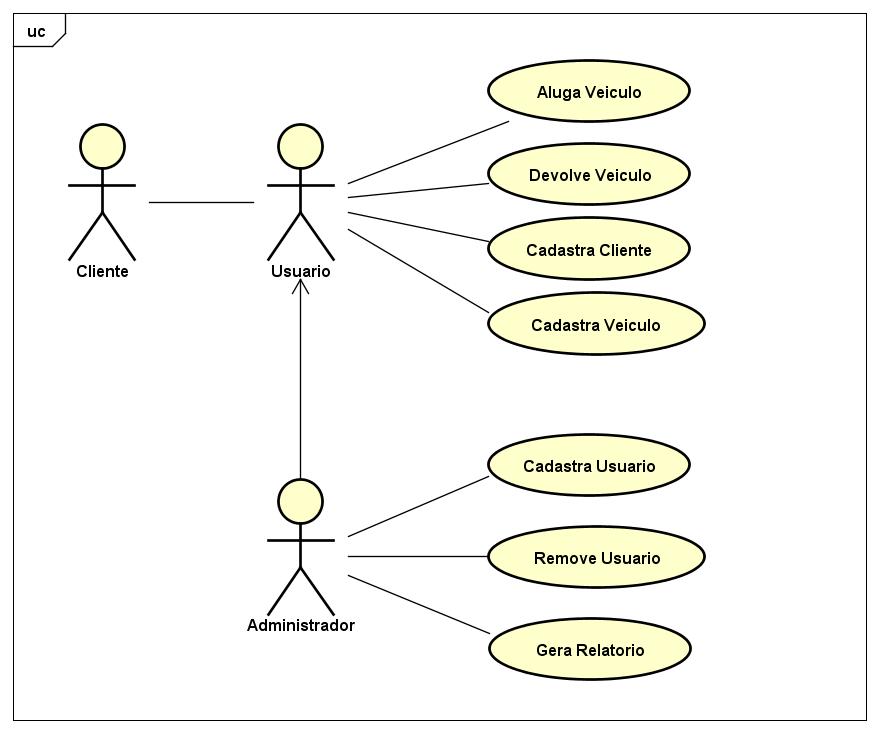
**Figura 7: Diagrama de Casos de Uso.**



Fonte: Autoral.

**3.2.2. Diagrama de Caso de Uso**

Na figura 7, ilustramos o diagrama de caso de uso para o dos atores Cliente, Usuário e Administrador, ligados às suas respectivas funções.

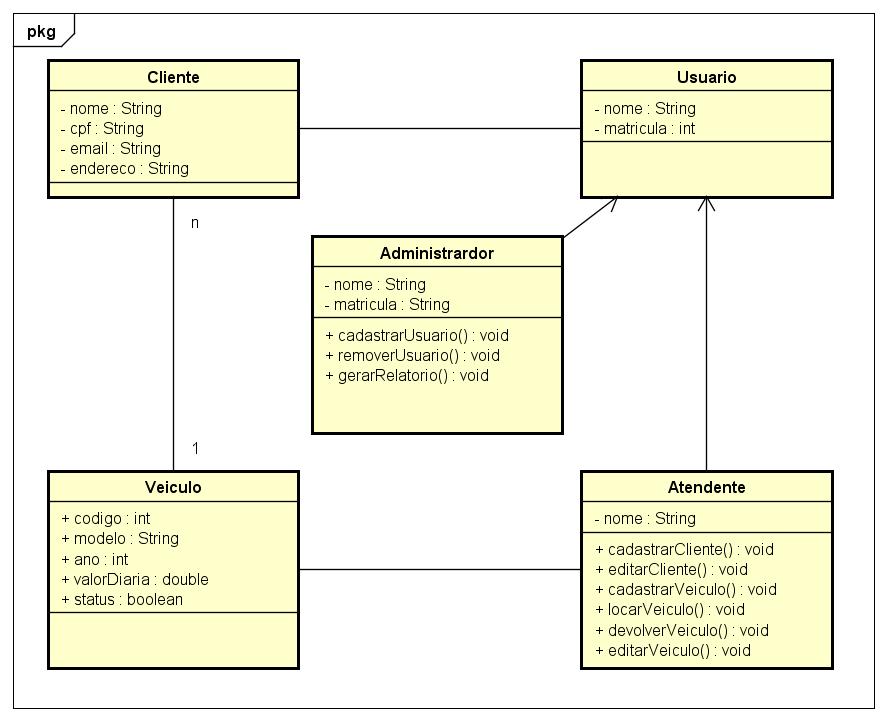
**Figura 7: Diagrama de Casos de Uso.**

Fonte: Autoral.

**3.2.3. Diagrama de Classe**

Para o cenário de nossa aplicação, foram identificadas incialmente 5 classes pricipais as quais possuem atributos e métodos próprios. E foram estabelecidas também relação de herança entre classes semelhantes conforme ilustra a figura 8.

Vale ressaltar que este diagrama foi concebido apenas como esboço inicial e que certamente sofrerá algumas modificações ao passo que novas funcionalidades forem sendo implementadas e as já existentes forem alteradas.

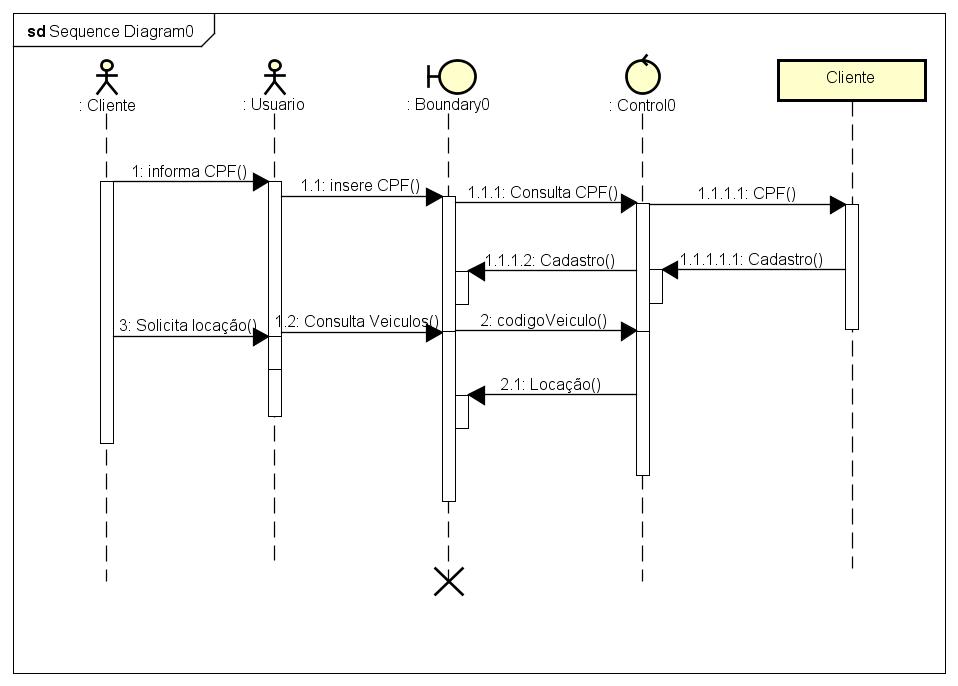
**Figura 8: Diagrama de Classes.** ****

Fonte: Autoral.

**3.2.4. Diagrama de Sequencia**

Seguindo o contexto de nossa aplicação, a figura 9 abaixo demonstra de forma sequêncial uma possível operação de locação de um veículo, ilustrando os atores e componentes envolvidos bem como as mensagens transmitidas nessas operações.

**Figura 9: Diagrama de Sequência.**

****

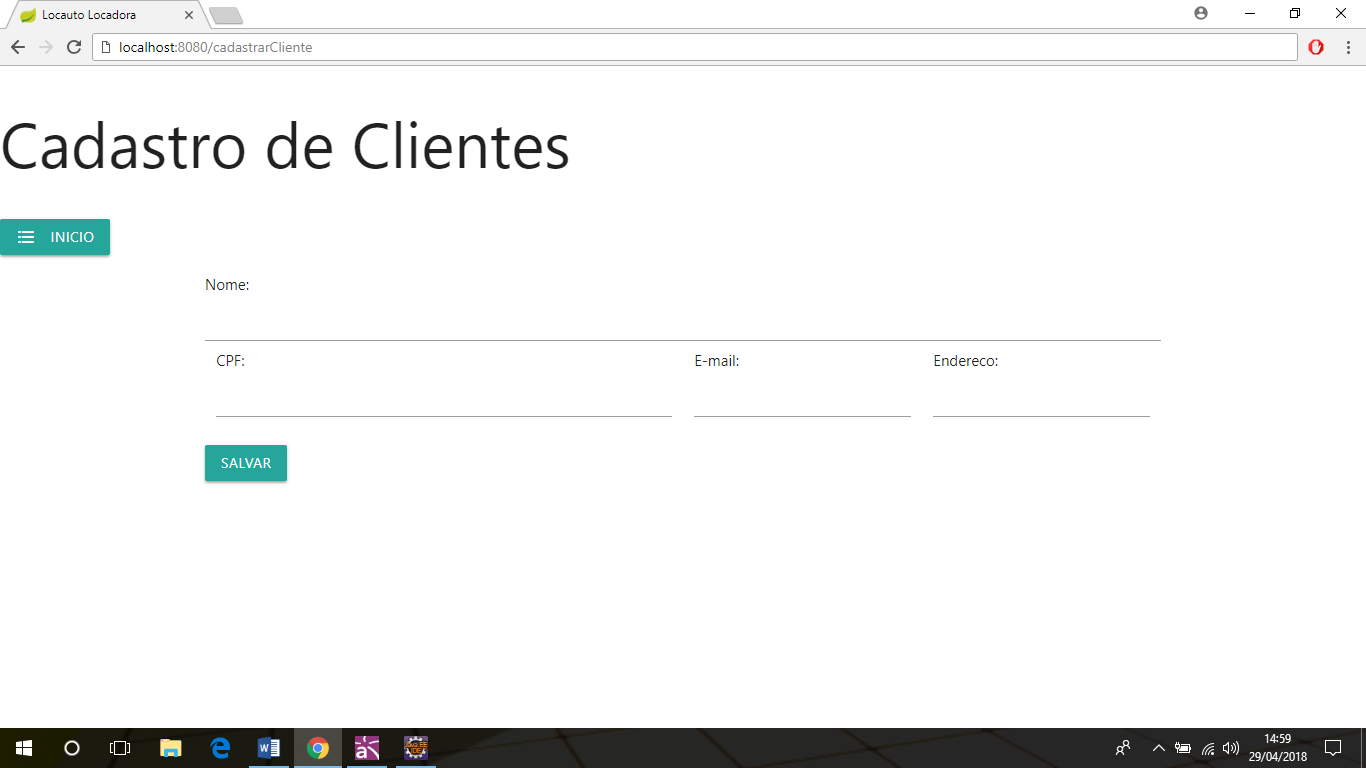
Fonte: Autoral.

**3.2.5. Protótipo**

Abaixo temos uma tela de amostra da possível pagina de cadastro de clientes.

Por se encontrar ainda em fase de desenvolvimento o design final poderá sofrer algumas alterações.

**Figura 10. Tela de Cadastro de Clientes.**



Fonte: Autoral.

**4. CONCLUSÃO**

Como vimos, o desenvolvimento de um software, mesmo que básico, exige uma que sejam seguidos uma série de passos visando acelerar as fases mais demoradas e prevenir possíveis erros que possam ser cometidos durante as etapas desse processo. Para sistemas mais complexos e cheios de funcionalidades, se faz necessário o uso exaustivo de todas as ferramentas necessárias para que se obtenha ao final um software de alto padrão de qualidade e que atenda plenamente os fins para o qual foi idealizado e concebido.

Algumas partes deste artigo foram suprimidas momentaneamente, devido ao fato desta aplicação ainda se encontrar em fase inicial do seu desenvolvimento. Uma nova versão completa deverá ser lançada tão logo se tenha concluído o processo de codificação e testes e já se tenha um aplicativo funcional pronto para entrar em operação.

REFERÊNCIAS

FELIPE, Eduardo. **Principais conceitos da Programação Orientada a Objetos.** Disponível em: <https://www.devmedia.com.br/principais-conceitos-da-programacao-orientada-a-objetos/32285>. Acesso em: abr.2018.

MACHADO,Henrique, **Os 4 pilares da Programação Orientada a Objetos.**

Disponível em: <<https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264>>. Acesso em: abr.2018.

XAVIER, Denys William. **Interfaces**: Usando uma Interface em uma Classe.

Disponível em: <<http://www.tiexpert.net/programacao/java/interfaces.php>>. Acesso em: abr.2018.

MARTINS, Vidal. **O Processo unificado de desenvolvimento de Software**. Disponível em: <http://www.batebyte.pr.gov.br/modules/conteudo/conteudo.php>. Acesso em: abr.2018.

RICARTE, Ivan Luiz Marques. **O que é uma classe.** Disponível em: <http://www.dca.fee.unicamp.br/cursos/PooJava/classes/conceito.html>. Acesso em: abr.2018.

NOGUEIRA, Admilson. **UML – Unified Modeling Language – Introdução e Historico.** Disponível em: <http://www.linhadecodigo.com.br/artigo/763/uml-unified-modeling-language-introducao-e-historico.aspx>. Acesso em: abr.2018.

MADEIRA, Marcelo. **Entendendo o Diagrama de Casos de Uso.** Disponível em:

<https://celodemelo.wordpress.com/2007/03/17/entendedo-o-diagrama-de-casos-de-uso/>. Acesso em: abr.2018.

CÉSAR,Paulo, **Utilizando UML: Diagrama de Classes.** Disponível em: < https://www.devmedia.com.br/diagrama-de-classes-uml/12251>. Acesso em: abr.2018.

IBM KNOWLEDGE CENTER. **Diagramas de Seqüência**. Disponível em: <https://www.ibm.com/support/knowledgecenter/pt-br/SSCLKU\_7.5.5/com.ibm.xtools.sequence.doc/topics/cseqd\_v.html>. Acesso em: abr.2018.

VENTURA, Plínio. **Entendendo o Diagrama de Atividades da UML**. Disponível em: <http://www.ateomomento.com.br/uml-diagrama-de-atividades/>. Acesso em: abr.2018.

JOSÉ, Renato. **Modelagem de sistemas através de UML: uma visão geral.** Disponível em: <https://www.devmedia.com.br/modelagem-de-sistemas-atraves-de-uml-uma-visao-geral/27913>. Acesso em: abr.2018.

RAMOS, Allan. **O que é MVC?** Explicando o MVC, um padrão de arquitetura para organizar sua aplicação. Disponível em: < https://tableless.com.br/mvc-afinal-e-o-que/>. Acesso em: abr.2018.