INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO

Campus São João da Boa Vista

Trabalho Final de Curso

4º ano – Curso Técnico em Informática

Prof. Breno Lisi Romano e Prof. Luiz Angelo Valota Francisco

UTILIZAÇÃO DA ARQUITETURA MVC NO DESENVOLVIMENTO DO PROJETO GERAÇÕES

Aluno: Bruno Gomes Fernandes

Prontuário: bv1620631

São João da Boa Vista – SP

2019

Resumo

O objetivo deste documento é realizar um estudo completo acerca da implementação e aplicação da arquitetura Model-View-Controller (MVC) no desenvolvimento de softwares, com enfoque para sua utilização na aplicação do Projeto Gerações, desenvolvida durante todo o ano de 2019 na disciplina de Projeto e Desenvolvimento de Sistemas (PDS) pelos alunos do 4º Ano do Curso Técnico Integrado em Informática do IFSP-SBV. Para tal, serão explanados conceitos básicos acerca de Programação Orientada a Objetos, Padrões de Projeto e do próprio MVC. Ademais, este documento também contempla a uma completa exposição e análise da aplicação e do funcionamento de tais conceitos no âmbito do projeto desenvolvido, através de códigos construídos utilizando a Linguagem de Programação PHP.

Palavras-Chave: Programação Orientada a Objetos; Padrões de Projeto; Arquitetura MVC; Linguagem PHP;

Sumário

Índice de Ilustrações

Lista de Tabelas

1 Introdução

1.1 Contextualização / Motivação

Fundada no ano de 1909 sob a alcunha de "Escola de Aprendizes e Artífices" pelo então presidente da república Nilo Peçanha, a Rede Federal de Educação Profissional, Científica e Tecnológica possuí hoje presença em todos os estados brasileiros. Seus principais representantes atuais são os Institutos Federais de Educação, Ciência e Tecnologia, criados em dezembro de 2008 a partir dos Centros Federais de Educação Tecnológica (CEFETs) e outras instituições como as Unidades Descentralizadas de Ensino (UNEDs), e que hoje totalizam cerca de 644 campi em funcionamento, oferecendo cursos de qualificação e técnicos integrados ao ensino médio, além de cursos superiores de tecnologia, bacharelados e licenciaturas [1][2].

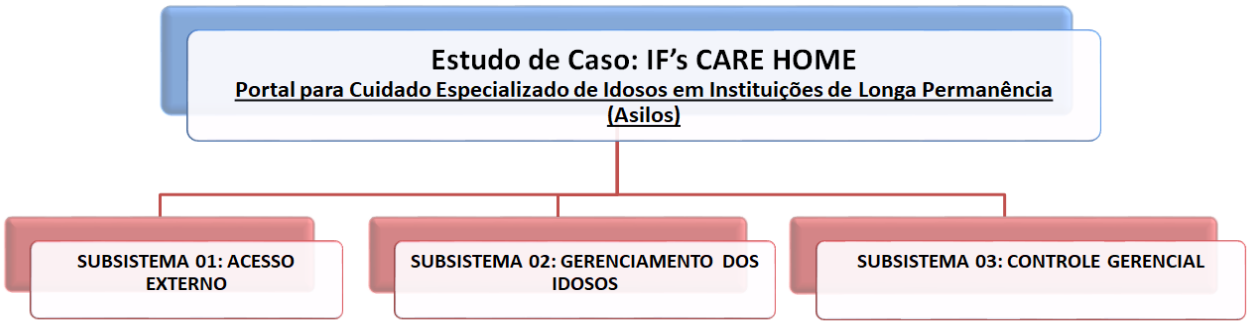
Uma destas 644 unidades educacionais está localizada na cidade de São João da Boa Vista, interior de São Paulo, município que conta atualmente com uma população de aproximadamente 91.211 pessoas, segundo dados coletados pelo IBGE no início do ano de 2019. Fundado em janeiro de 2007, o campus São João da Boa Vista do Instituto Federal de São Paulo (IFSP) conta hoje com cursos de múltiplos níveis: oito cursos técnicos, integrados ou concomitantes/subsequentes ao ensino médio, cursos superiores – dentre bacharelados, engenharias, licenciaturas e cursos de tecnologia – além de programas de pós-graduação nas áreas de informática, educação e humanidades. Reconhecido pela sua excelência no oferecimento de um ensino público de altíssima qualidade, sobretudo devido à sua moderna infraestrutura e corpo docente altamente qualificado, o IFSP-SBV tornou-se um centro de referência na capacitação de jovens e adultos das mais variadas faixas etárias em nossa região [3][4].

Dentre os cursos oferecidos pela instituição, podemos destacar o Curso Técnico Integrado em Informática pela alta adesão de jovens da região de São João da Boa Vista, que além de oferecer um ensino médio de altíssima qualidade, capacita profissionalmente técnicos da área de informática para o levantamento e análise de dados, diagnóstico e manutenção de sistemas computacionais, além do desenvolvimento de aplicações digitais (*softwares*). Para tal, o curso oferece em sua grade curricular disciplinas de Manutenção de Computadores, Lógica, Linguagens de Programação, Banco de Dados, Redes de Computadores e Engenharia de Software. No último ano, a disciplina de Prática de Desenvolvimento de Sistemas (PDS) oferece aos alunos a oportunidade – e desafio – de vivenciar um pouco do cotidiano de um profissional da área de informática, exigindo deles a aplicação de todo o conhecimento adquirido durante os quatro anos de curso, além de propiciar um ambiente de trabalho em equipe similar ao de uma empresa de desenvolvimento de software. Todos os anos, um novo projeto de software é desenvolvido com o objetivo de atender uma necessidade da comunidade local, atribuindo à disciplina um valor de cunho social [5][6].

Pensando no bem-estar dos idosos do munícipio de São João da Boa Vista – considerada a melhor cidade brasileira entre 50 e 100 mil habitantes para os indivíduos da terceira idade, segundo pesquisa desenvolvida pela Fundação Getúlio Vargas (FGV) em 2017 – surge a ideia da elaboração, em 2019, de um projeto que atenda à população idosa da cidade, mais precisamente daqueles que se encontram instalados em Instituições de Longa Permanência. Nasce então o projeto Gerações, cujo principal objetivo é a criação de um portal para cuidado especializado de idosos nestas instituições, conhecidas popularmente como asilos [7][8].

Tendo em vista a complexidade da aplicação a ser desenvolvida, é realizada a divisão do sistema idealizado em 3 Subsistemas: O Subsistema 01, de Acesso Externo, que trata da conexão dos diferentes tipos de usuários (enfermeiros, médicos e sobretudo os responsáveis pelos idosos, principais interessados pelas funcionalidades oferecidas pelo subsistema) com a aplicação, o Subsistema 02, responsável pelo Gerenciamento dos Idosos (cuidados diários, nutrição, análises clínicas etc.), e o Subsistema 03, que efetua o controle gerencial e administrativo da Instituição de Longa Permanência [8].

Figura 1. Esquematização dos Subsistemas pertencentes ao projeto Gerações [8]



Além da divisão em subsistemas, o projeto Gerações foi também dividido em módulos – equipes de 6 a 7 pessoas, divididas de acordo com funções pré-estabelecidas no projeto, que devem ser desempenhadas durante todo o desenvolvimento do projeto, da documentação inicial e especificação de requisitos à codificação e implementação do sistema propriamente dito. Cada um dos 9 módulos conta com 2 Analistas de Sistemas/Testadores – responsáveis, inicialmente, pela especificação e documentação do sistema e, a partir da fase de implementação do projeto, por realizar os testes das funcionalidades – 2 DBAs – responsáveis pela especificação e desenvolvimento do banco de dados da aplicação – e de 2 a 3 Desenvolvedores – responsáveis pela implementação da identidade visual gráfica do projeto (*Logo* e *Template*) e suas funcionalidades, utilizando para isso a documentação elaborada pelos analistas e a base de dados desenvolvida pelos DBAs [8].

Figura 2. Esquematização dos Módulos 04 ao 07 e seus Respectivos Macrorequisitos [8]

Considerando-se os trabalhos a serem desempenhados pela equipe de desenvolvedores do projeto durante o ano, têm-se que, além da definição inicial da identidade visual do projeto, é de sua responsabilidade a confecção de códigos-fonte que estruturam as páginas *web* da aplicação e suas respectivas interfaces gráficas (a aparência das telas para o usuário) – o chamado desenvolvimento *front-end*, que faz uso de tecnologias como HTML, CSS, Javascript e *frameworks* como o Bootstrap – e a implementação das regras de negócio / requisitos do projeto, no chamado desenvolvimento *back-end,* parte da codificação que, como seu próprio nome indica, fica "por trás" da aplicação, ou seja, não está visível para o usuário, e que é construída utilizando linguagens de programação, como por exemplo, no caso de nosso projeto, a linguagem PHP, e que dá vida às funcionalidades requisitadas pelo cliente de um projeto de software.

1.2 Objetivos Gerais

Considerando-se o ambiente de desenvolvimento do projeto estabelecido no início do projeto, com a divisão da equipe de trabalho em módulos, conforme o descrito na seção anterior, verifica-se a necessidade da utilização de diretrizes e normas que orientem o trabalho em grupo a ser desempenhado durante todo o desenvolvimento da aplicação, sobretudo no que se diz respeito à organização dos códigos-fonte em um padrão a ser seguido por toda a equipe de desenvolvedores. É diante de tal busca pela padronização do trabalho de desenvolvimento que a utilização da arquitetura *Model-View-Controller (MVC)* demonstra-se vital para o sucesso dos trabalhados no projeto Gerações.

Portanto, o objetivo geral deste trabalho compreende a especificação da concepção, implementação e funcionamento da arquitetura MVC e no processo de desenvolvimento da aplicação *web* do Projeto Gerações.

1.3 Objetivos Específicos

A partir do Objetivo Geral citado na seção anterior, destacam-se como objetivos específicos deste trabalho:

- Pesquisar e apresentar conceitos básicos de Linguagens e Paradigmas de Programação, com foco especial à Programação Orientada a Objetos (POO);
- Introduzir o PHP, contemplando sua história e conceitos básicos;
- Introduzir o conceito de Padrão de Projeto e a arquitetura MVC de desenvolvimento de software, incluindo suas origens e principais conceitos;
- Expor e analisar a aplicação dos conceitos de POO no Projeto Gerações através da linguagem PHP;
- Especificar e realizar uma explanação acerca dos três tipos de classes principais que fazem parte de uma aplicação desenvolvida seguindo o padrão MVC: *Models, Views* e *Controllers;*
- Apresentar e explanar a implementação de tal padrão de desenvolvimento de softwares ao Projeto Gerações;

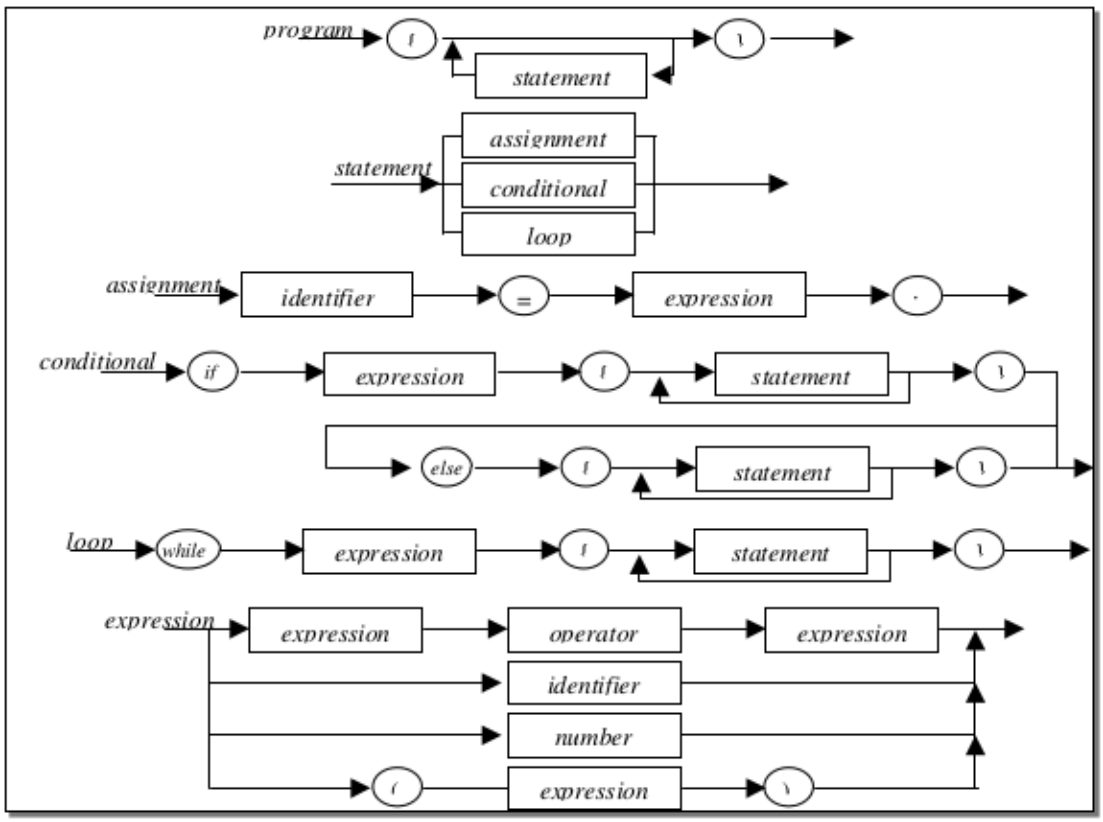
2 Desenvolvimento

2.1 Levantamento Bibliográfico

2.1.1 Introdução à Linguagens de Programação

Dentro do escopo de um Sistema Computacional, isto é, de uma unidade computacional composta por circuitos elétricos e dispositivos eletrônicos (*hardware*) utilizados para o processamento de informações geradas por agentes lógicos, como por exemplo aplicativos e programas de computador (*software*), podemos entender uma Linguagem de Programação como uma espécie de "ponte" entre o usuário e o computador – é, em suma, a maneira na qual podemos comunicar ao computador o que ele deve fazer. Comparando as Linguagens de Programação com tipos de linguagem observados na vida real, como nossa própria língua, podemos estabelecer uma série de semelhanças: Assim como em nossa língua, as Linguagens de Programação também possuem um conjunto de palavras (vocabulário), que chamamos de instruções ou comandos. Além disso, também possuem uma sintaxe, um conjunto de regras que define como podemos organizar tais palavras na construção do que chamamos de código, a fim de gerar instruções válidas que podem ser executadas pelo computador. Em suma, utilizamos tais linguagens para a construção de programas de computador destinados à solução de problemas específicos do mundo real [9][10].

Figura 3. Exemplo de Diagrama de Sintaxe de Programação [11]



2.1.2 Níveis de Linguagens de Programação

Podemos classificar as Linguagens de Programação existentes seguindo diversos conceitos, sendo que um dos mais comuns diz respeito ao nível da linguagem, isto é, de sua proximidade com a linguagem da máquina (no caso, o computador). Neste sentido, podemos classificar, de maneira geral, as linguagens de programação em 3 grupos: Linguagens de Baixo Nível, Linguagem Assembly e Linguagens de Alto Nível [9].

Linguagens de Baixo Nível (também conhecidas como Linguagens de Máquina) são aquelas que, como sugerido pela sua própria nomenclatura, mais se aproximam das instruções executadas pelos computadores, e por isso geralmente seus códigos são compostos por sequências de números, que podem ser feitas em notação binária ou hexadecimal. Na imagem abaixo, temos um exemplo de código de linguagem de máquina escrito em notação binária – podemos observar uma série de números 0 e 1, que denotam, para a máquina, ligado ou desligado. Para cada dígito exibido abaixo, atribuímos o nome de *bit*, abreviatura de *Binary Digit*. Para cada conjunto de oito bits, atribuímos o nome de *byte* [9].

Figura 4. Exemplo de Código em Linguagem de Máquina

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

A Linguagem Assembly é composta por abreviações de expressões da língua inglesa, facilitando a compreensão humana, ao mesmo tempo que ainda é uma linguagem facilmente traduzida para a linguagem de máquina, uma vez que é um tipo de linguagem que pode ser executado de maneira direta pelas unidades de processamento dos computadores [9].

Figura 5. Exemplo de Código em Linguagem Assembly

```
mov ax,cs
mov ds,ax
mov ah,9
mov dx, offset Hello
int 21h
xor ax,ax
int 21h

Hello:
  db "Hello World!",13,10,"$"
```

Já as Linguagens de Alto Nível são aquelas que estamos mais acostumados a ver em nosso cotidiano, possuindo instruções e comandos representados por palavras que se aproximam mais da linguagem humana, especialmente da língua inglesa, além de uma sintaxe que pode ser facilmente compreendida por nós. Para que códigos neste tipo de linguagem sejam compreendidos e devidamente executados pela máquina, porém, é necessário que ocorra um processo de tradução, realizado pelos chamados compiladores [9].

Figura 6. Exemplo de Código em uma Linguagem de Alto Nível

```
int main()
{
    int n1, n2, soma;

    printf("Escreva o primeiro número:\n");
        scanf("%d", &n1);

    printf("Escreva o segundo número: \n");
        scanf("%d", &n2);

    soma = n1 + n2;

    printf("O valor da soma é %d", soma);

}
```

2.1.3 Paradigmas de Programação

De maneira geral, podemos dizer que um Paradigma de Programação faz referência ao *approach* utilizado para visualizar e solucionar um problema utilizando uma linguagem de programação, baseado em um ponto de vista do mundo real. Neste sentido, a história das linguagens de programação representa uma evolução de paradigmas baseada nas necessidades e na orientação dos processos de desenvolvimento de cada período. Nesta seção, citaremos de maneira breve alguns dos paradigmas de programação que obtiveram maior destaque na história [9][12].

2.1.3.1 Programação Estruturada

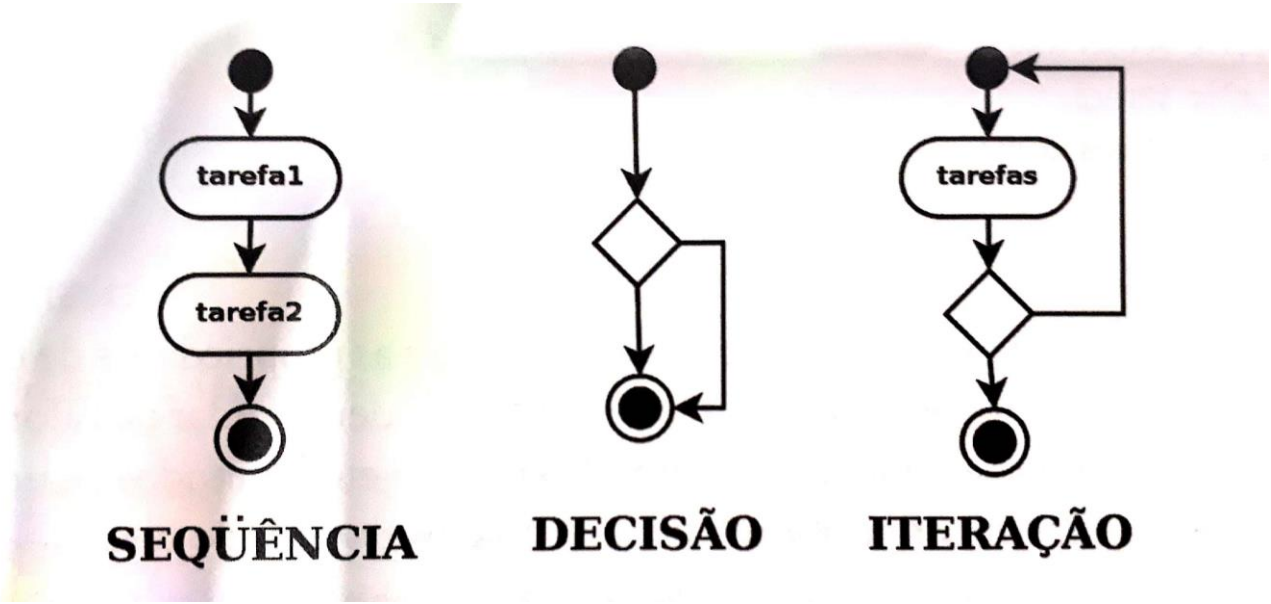
Dentro do que hoje chamamos de Programação Estruturada, podemos citar dois paradigmas que foram extremamente importantes para o desenvolvimento das linguagens de programação e dos processos de desenvolvimento de software. O primeiro deles a surgir, conhecido como Paradigma Imperativo ou Procedural, demonstra como principal característica o entendimento e a divisão de um programa de computador em procedimentos, isto é, passos/etapas a serem executados em sequência

durante a execução do programa em si. Em cada um destes procedimentos, que podem ser sub-rotinas ou funções, estão contidos trechos de códigos responsáveis pela execução de instruções específicas dentro do escopo do código.

Com o passar dos anos, verifica-se na história da programação o crescimento de uma orientação para a produção de códigos que possuíssem uma legibilidade mais simples do que aqueles produzidos pelo paradigma procedural. É neste contexto que surge outro paradigma de programação estruturada, o paradigma de Programação Modular, que visa cumprir tal objetivo através da utilização de funções, sub-rotinas e variáveis agrupados em diferentes módulos, isto é, divisões do código do programa realizadas conforme a sua função dentro do escopo geral. Ou seja, no paradigma de Programação Modular dividimos, portanto, funções, sub-rotinas e variáveis para que cada uma dessas divisões ou módulos desempenhe um papel específico dentro do código da aplicação [9][12].

De maneira geral, todos os programas criados utilizando os paradigmas de programação estruturada citados acima podem ser compreendidos através de 3 elementos chave, estruturas básicas de controle que interconectam os procedimentos (funções e sub-rotinas presentes no código) e que garantem a correta execução do código como uma unidade e, desta forma, o funcionamento correto do programa: Sequência – a sequência de passos durante a execução do programa –, Decisão – o processo de escolha de fluxos alternativos baseada em estruturas condicionais – e Iteração – a capacidade de realizar a repetição da execução de trechos do código quando necessário [9][12][13].

Figura 7. Programação Estruturada: Estruturas Básicas de Controle [13]



2.1.3.2 Programação Orientada a Objetos

Avançando o tempo para os dias atuais, podemos destacar o Paradigma de Programação Orientada a Objetos – um dos últimos a surgir e se popularizar – como um dos, se não o mais utilizado

entre os desenvolvedores de *software* atualmente. Tal cenário surge diante da extrema popularidade que linguagens de programação que utilizam tal paradigma ganharam nas últimas décadas, como por exemplo as linguagens Java, PHP, C#, C++, .NET, Python, dentre muitas outras [12].

Considerado o paradigma sucessor da programação estruturada, a Programação Orientada a Objetos (POO) tem como principal proposta uma maior aproximação dos processos de desenvolvimento de *softwares* da realidade, tornando tais processos muito mais intuitivos. Neste sentido, o que a Orientação a Objetos faz é, segundo Gilmore (2011, p. 132), deslocar o foco dos eventos dos procedimentos do programa para as entidades da vida real que ele modela. De maneira geral, portanto, os programas desenvolvidos utilizando POO têm seu fluxo principal orientado pelos relacionamentos entre tais entidades, chamadas de objetos [13][14][15].

2.1.4 Conceitos de Orientação a Objetos

Para compreendermos melhor a programação orientada a objetos e o seu papel de extrema importância para o padrão de desenvolvimento utilizado em nosso projeto, é necessário que compreendamos primeiramente alguns de seus conceitos-chave. Introduziremos nesta seção, portanto, tais conceitos.

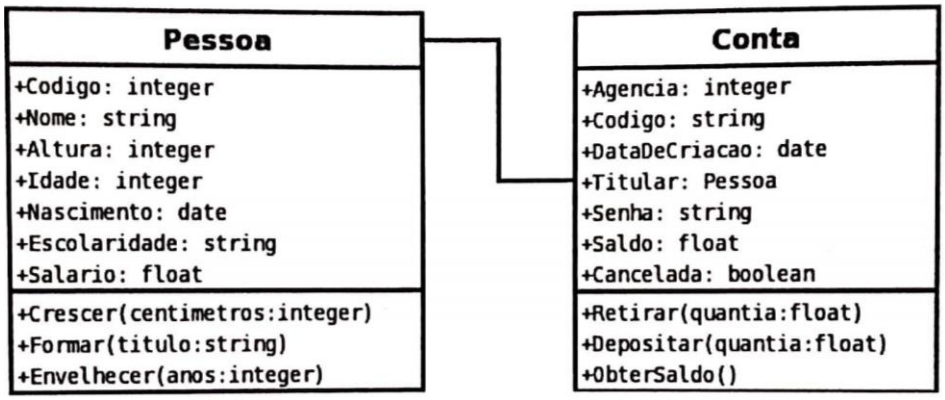
2.1.4.1 Classe

Dentro de POO, uma classe funciona como um "molde" que provê a estrutura básica para que possamos fazer a definição de objetos pertencentes à tal classe. Isso envolve, em suma, definir suas características (atributos/variáveis) e comportamentos (métodos/funções). Neste sentido, conforme elencado por Gilmore (2011, p. 133), "classes são destinadas para representar aqueles itens da vida real que você gostaria de manipular dentro de uma aplicação". Se quiséssemos desenvolver, por exemplo, um *software* para uma escola, deveríamos utilizar classes como Alunos, Professores, Funcionários, Turmas, Salas de Aula, Materiais, Livros etc. Por esse motivo, a especificação das classes constitui uma etapa muito importante da concepção do *software*, sendo uma das primeiras etapas de todo processo de desenvolvimento [13][14][15].

Em UML – abreviatura para Linguagem de Modelagem Unificada – uma classe é ilustrada como uma caixa retangular subdividida em outros três retângulos – o primeiro (superior) contém o nome da classe em questão, o segundo lista seus atributos (características), e no terceiro são elencados os seus métodos/operações (comportamentos). Nas imagens abaixo podemos observar dois exemplos de classes em notação UML – a classe Pessoa e a classe Conta. Como podemos observar, a classe Pessoa possui os atributos código, nome, altura, idade, data de nascimento, escolaridade e salário. Além disso, possuí também os métodos Crescer, Formar e Envelhecer – que são os seus comportamentos, as ações que podem ser desempenhadas por objetos da classe pessoa. Já a classe

conta possuí como seus atributos a agência, o código, a sua data de criação, o nome do titular, sua senha, o saldo disponível e seu *status*, por meio do atributo cancelada, que indica se está ainda é válida ou foi cancelada. Já os métodos / comportamentos da classe conta são *Retirar, Depositar e Obter Saldo* [13][14][15].

Figura 8. Notação de Classe em UML [13]



2.1.4.2 Objeto

Conforme já foi citado acima, os objetos constituem, como a própria designação do paradigma nos sugere, o foco principal de uma aplicação desenvolvida utilizando POO – é a partir deles que efetuaremos a construção completa de um sistema utilizando tal técnica, desde as etapas iniciais de modelagem e especificação de requisitos, até a implementação e os testes posteriores da aplicação desenvolvida. Portanto, é de suma importância que compreendamos de maneira correta o significado de tal conceito.

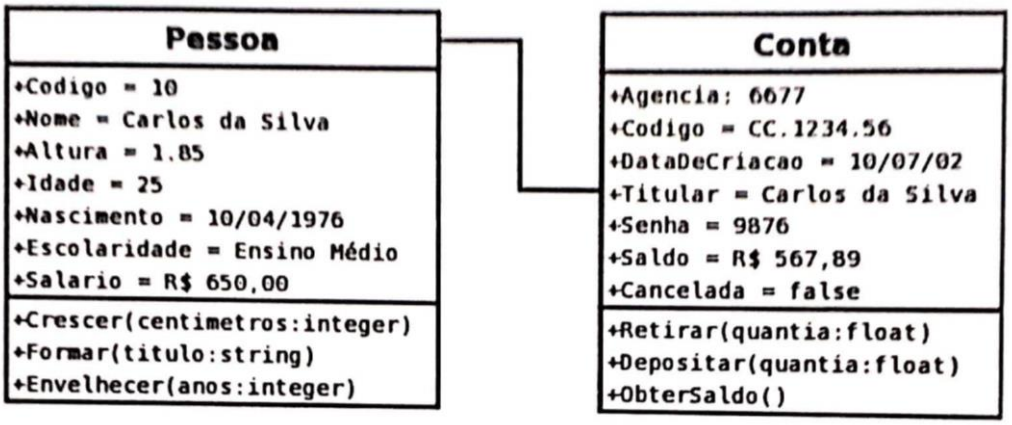
Segundo Dall'Oglio (2009, p. 93), podemos compreender um objeto como uma estrutura gerada dinamicamente, em um processo que utiliza como base uma classe previamente criada. Chamamos tal processo – realizado pelos chamados métodos construtores, que veremos com mais detalhes posteriormente – de *instanciação de um objeto*. Neste sentido, podemos dizer que o objeto nada mais é do que a instância de uma classe, importando, desta forma, sua estrutura – ou seja, todos as características (atributos) e comportamentos (métodos) nela definidos [13].

Pensando nos exemplos de classe dados acima, podemos imaginar possíveis objetos a elas pertencentes e veremos que, apesar de todos terem sido concebidos a partir de uma mesma estrutura pré-determinada pelas suas classes, que serviram como molde para sua criação, observaremos que seus atributos podem assumir valores distintos. Tal distinção nos valores dos atributos é o que diferencia os diferentes objetos pertencentes a uma mesma classe. Se pensarmos na classe pessoa, por exemplo, podemos ter o objeto João, de idade 11 anos e 1,50m de altura, ao mesmo tempo que também podemos construir o objeto Maria, que possuí idade 55 e 1,70m de altura. Neste sentido, é

possível perceber que podemos modelar e instanciar objetos completamente distintos que pertencem à mesma classe – neste caso, a classe Pessoa [13].

A imagem abaixo ilustra a instanciação de objetos das duas classes exemplificadas na sessão anterior – Pessoa e Conta. Como podemos ver, cada um dos atributos de ambas as classes recebe valores que essencialmente irão tornar o objeto em questão único [13].

Figura 9. Exemplos de Objetos - Instanciações das Classes Pessoa e Conta [13]



2.1.4.3 Encapsulamento

Juntamente com Abstração, Herança e Polimorfismo, o Encapsulamento constitui um dos quatro pilares básicos de toda e qualquer linguagem de programação orientada a objetos, constituindo, portanto, um conceito-chave para que possamos compreender e explorar de maneira eficiente e correta as técnicas de POO.

Sintes (2002, p. 22) define o Encapsulamento como “a característica da OO de ocultar partes independentes da implementação”. O processo de encapsular consiste em, basicamente, “esconder” a implementação de parte do software de seu mundo exterior – ou seja, de partes externas deste mesmo software –, sem que, entretanto, a funcionalidade exercida por tal trecho de código se torne indisponível para a utilização de tais partes externas. De maneira geral, conforme elencado por Dall’Oglio (2009, p. 107), o encapsulamento é responsável por prover uma proteção de acesso aos elementos internos de um objeto (ou seja, seus atributos e métodos), garantindo que tais elementos não sejam acessados diretamente por partes do software ou entidades externas ao seu escopo [13][15].

Essencialmente, quando realizamos o encapsulamento de uma classe, por exemplo, o que estamos fazendo é afastar o seu funcionamento interno do usuário, fornecendo-o apenas uma interface conhecida para a utilização das funcionalidades de tal classe. Estabelecendo uma analogia com nosso cotidiano, isso ocorre de maneira similar à utilização de um controle remoto de televisão: a maioria de nós não possuí um entendimento completo acerca de seu funcionamento, porém isso não nos impede de utilizar suas funcionalidades (os botões). Quando apertamos um botão, nós sabemos que

função do controle estamos utilizando, mas não sabemos o que ocorre por trás para que tal função seja executada – o envio de mensagens codificadas em luz infravermelha, que pisca enviando pulsos que compõem o código binário responsável pela execução da função na televisão. Neste caso, podemos dizer que o controle remoto é a nossa interface [14][15].

Em um programa desenvolvido com orientação a objetos, podemos dizer que a interface é composta pelas chamadas dos métodos, enquanto tudo que não as constitui pode ser considerado como implementação (os detalhes de funcionamento que não interessam ao usuário que as utiliza). Caso a interface seja pública, teremos apenas os seus métodos públicos disponíveis para utilização do usuário – isso significa que métodos definidos com outros tipos de visibilidade não estarão disponíveis em tal interface. É essencialmente neste cenário que reside o principal recurso de POO que podemos utilizar para efetuar o encapsulamento de uma classe: a definição da visibilidade de seus métodos e atributos [13][15].

De maneira geral, a visibilidade de um método ou atributo define como e por quem (por quais entidades externas) tais elementos podem ser acessados. Em toda linguagem de programação orientada a objetos, existem três formas básicas de visibilidade: *Public* (público), *Private* (privado) e *Protected* (protegido). Abaixo construímos uma tabela com a respectiva descrição de cada um desses tipos de visibilidade:

Tabela 1. Encapsulamento: Tipos de visibilidade em POO [13][15]

Tipo de Visibilidade	Descrição
Public (+)	O acesso a elementos declarados com visibilidade pública pode ser realizado a partir de qualquer entidade do escopo do programa que faz uso de tal classe. Seu símbolo em UML é o (+).
Private (-)	O acesso a elementos declarados com visibilidade privada pode ser feito apenas a partir da própria classe que os contêm. O acesso é vedado, desta forma, para qualquer entidade externa à classe de origem – na prática, apenas instâncias (objetos) de tal classe detêm tal acesso. Seu símbolo em UML é o (-).
Protected (#)	O acesso a elementos declarados com visibilidade *protected* é garantindo não só para instâncias (objetos) da classe de origem dos elementos, mas também a partir de instâncias de classes descendentes. Seu símbolo em UML é o (#).

Além de promover a segurança de acesso à elementos internos das classes, diminuindo muito a probabilidade de erros causados por relacionamentos inadequados entre objetos, o encapsulamento também permite sua independência e transparência em alterações neles realizados [15].

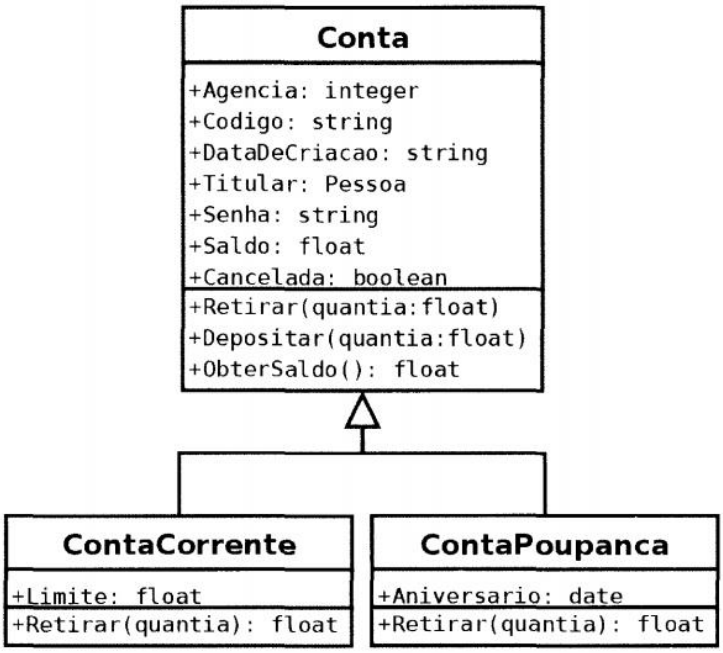
2.1.4.2 Herança

Quando falamos em Orientação a Objetos, uma das principais vantagens que podemos elencar, além de uma ótima legibilidade e organização do código, é o alto poder de reutilização deste que programas desenvolvidos conforme tal paradigma possuem, conferindo uma grande praticidade e agilidade para o processo de implementação do código. Grande parte deste "poder" se deve a outro de seus pilares básicos: o mecanismo de herança [13].

Segundo Sintes (2002, p.72), podemos definir Herança em POO como a capacidade de basear a definição de uma nova classe em outra classe já existente, de maneira que a nova classe herde todos os atributos, métodos e implementações da classe em que sua definição foi baseada, de maneira similar à transmissão das características que ocorre geração após geração em uma árvore genealógica de seres humanos [13][15].

O diagrama UML abaixo representa um relacionamento de herança entre 3 classes: *Conta*, *ContaCorrente* e *ContaPoupança*. Como podemos observar, as classes *ContaCorrente* e *ContaPoupança* são descendentes da classe *Conta* – dizemos então que elas são as classes-filha, enquanto a classe *Conta* é a classe-mãe. Neste relacionamento, as classes-filha herdam todos os atributos e métodos da classe-mãe, porém, como é possível observar, cada uma delas apresenta um novo atributo em relação à sua classe progenitora: A classe *ContaCorrente* possuí o atributo *Limite*, enquanto na *ContaPoupança* temos o atributo *Aniversario*. Além disso, em ambas as classes podemos notar a repetição do método *Retirar()* da classe-mãe – neste caso, o que ocorre é a chamada sobescrita de método (*overriding*), em que as classes-filha modificam o comportamento do método presente na classe originária [13].

Figura 10. Representação Gráfica (UML) de um relacionamento de Herança [13]



De maneira geral, o mecanismo de Herança é extremamente útil para POO pois possibilita, conforme já citado anteriormente, a reutilização do código, uma vez que dispensa a necessidade da reescrita de trechos de código para a criação de novas classes. Ao invés disso, podemos apenas utilizar o mecanismo de herança para hedar as características básicas de uma classe e realizar as adaptações necessárias, provendo novos atributos e métodos e/ou realizando a sobescrita destes [13][15].

2.1.4.3 Polimorfismo

A palavra *Polimorfismo* possuí como definição, em seu sentido mais literal, "a qualidade ou estado de ser capaz de assumir diferentes formas". Em POO, segundo Dall'Oglio (2009, p.101), Polimorfismo nada mais é do que "o princípio que permite que classes derivadas de uma mesma superclasse tenham métodos iguais, mas comportamentos diferentes, redefinidos em cada uma das classes-filha" [13].

Conforme ressaltado por Sintes (2002, p. 122), o Polimorfismo é, muito provavelmente, o mecanismo que mais contribuí com a eficácia das linguagens de programação orientadas a objetos. Construído à partir dos dois pilares anteriores, é através do Polimorfismo que podemos utilizar um único nome para expressar diversos comportamentos diferentes [15].

Um exemplo de Polimorfismo se encontra no caso ilustrado pela Figura 10 na seção anterior, em que o método *Retirar()* está presente em todas as classes, porém possuí uma implementação que resulta em comportamentos diferentes em cada uma delas – para cada uma delas, uma verificação diferente pode ser realizada no momento da execução do método. Em contas correntes, por exemplo, o titular pode sacar mais dinheiro do que o disponível em seu saldo, utilizando o cheque especial, enquanto em contas poupança o valor do saque não pode ser maior do que o saldo atual [13][15].

2.1.4.4 Abstração

Quando pensamos na palavra Abstração, consideramos como seu significado o ato de isolar mentalmente um objeto de seu contexto, de maneira a nos concentrarmos exclusivamente em seus aspectos mais essenciais. Em POO, efetuamos o processo de abstração através da definição de classes abstratas – estruturas basais que concentram os aspectos mais essenciais, sem riquezas de detalhes, que serão utilizadas mais tarde para a construção de estruturas bem-definidas, em que a riqueza de detalhes é extremamente maior. Por esse motivo, classes abstratas não podem ser instanciadas – apenas suas classes-filhas poderão. No exemplo da Figura 10, podemos dizer que nenhum titular de contas em um banco poderá ter uma *Conta,* pois é esta é uma classe abstrata. Apenas objetos das classes *ContaPoupança* e *ContaCorrente* poderão, portanto, ser instanciados [13][15].

Classes abstratas também podem possuir métodos abstratos – métodos vazios que são apenas uma espécie de “assinatura”. Tais métodos devem necessariamente ser implementados nas classes-filha da classe abstrata em questão. Neste sentido, o método *Retirar()* da classe *Conta* exemplificada na Figura 10 pode ser considerado abstrato, e cada uma de suas classes-filha deverá conter uma implementação para tal método – tal implementação pode ser diferente em cada uma delas, seguindo o princípio do polimorfismo, conforme abordamos na seção anterior [13].

2.1.5 Introdução ao PHP

Concebida em meados do outono de 1994 por Rasmus Lerdorf, o PHP é um exemplo clássico de um projeto de linguagem de programação *open source* criada para suprir necessidades de um desenvolvedor e que, rapidamente, passaria a ser aperfeiçoada rapidamente para suprir as necessidades de uma comunidade a cada dia mais crescente – comunidade esta que possuí papel ativo em tal processo de aperfeiçoamento e modernização da linguagem. O que era inicialmente um projeto pessoal de um programador solitário – um script Perl/CGI utilizado para registrar o número de acessos ao seu currículo online – viria a se tornar em pouquíssimo tempo uma das linguagens de programação mais populares no âmbito do desenvolvimento web, dotada de inúmeros recursos que, juntos, a tornam uma linguagem única [13][14].

Figura 11. Logo da Linguagem PHP

Dentre os marcos mais importantes da história do PHP, podem sem destacados [13][14]:

- **1997:** Lançamento do PHP 2.0, desenvolvido em C e que já contava com recursos como a conversão de dados inseridos em formulários HTML para variáveis – o que justifica seu nome na época: Personal Home Page/Form Interpreter – além de uma grande quantidade de melhorias desenvolvidas em conjunto com uma comunidade de programadores.
- **1998:** Lançamento do PHP 3.0, versão que continha uma ferramenta de parsing (conversão do código PHP para HTML) mais eficiente, e que iniciaria o “boom” do PHP – em seu lançamento, o PHP 3.0 já contava com cerca de 50000 usuários ativos.

- **2000:** Lançamento do PHP 4.0, cuja principal mudança residia em uma reestruturação total de seu núcleo, agora chamado de *Zend Engine*. A versão 4 do PHP trouxe como principais melhorias e adições um gerenciamento de recursos aprimorado, suporte a POO, recursos de gerenciamento de sessão, criptografia, dentre outros. Na ocasião, cerca de 20% dos domínios da internet já utilizavam o PHP.
- **2004:** Lançamento do PHP 5.0, versão considerada um divisor de águas na história do PHP. Suas principais features podem ser resumidas à melhorias gigantescas no suporte à Orientação a Objetos, introdução de um sistema de gerenciamento de erros com try/catch, adição de um suporte avançado a XML e Web Services, além do suporte nativo ao servidor de banco de dados compacto SQLite.
- **2015:** Lançamento do PHP 7.0, cuja principal melhoria reside no aumento de performance gigantesco em relação à sua versão anterior: o PHP 7 é cerca de nove vezes mais rápido do que o PHP 5. Também foram incluídos inúmeros recursos que tornam o PHP ainda mais prático e eficiente.

De maneira geral, conforme elencado por Gilmore (2011, p.5), as principais características que conferem ao PHP toda a sua popularidade e a preferência de muitos programadores em utilizá-lo em detrimento de outras linguagens de programação podem ser resumidas em "4 Ps": praticidade, poder, possibilidade e preço [14].

Concebida por Lerdorf com enfoque na praticidade, a linguagem PHP pode ser considerada prática devido à uma série de características, dentre as quais podemos destacar a ausência da necessidade de efetuar a definição explícita dos tipos das variáveis, o que, consequentemente, elimina a necessidade de fazer conversões de tipos de dados para outros e de realizar a destruição de variáveis. Além disso, a linguagem possui uma sintaxe simplificada e dispensa a necessidade da declaração de bibliotecas. De maneira geral, essas e outras características conferem ao PHP uma grande facilidade de aprendizagem para os programadores, permitindo, segundo Gilmore (2011, p.5), que seus usuários construam aplicações muito poderosas mesmo com o mínimo de conhecimento [14].

PHP também é considerada uma linguagem poderosa pois conta com mais de 180 bibliotecas que, em conjunto, disponibilizam ao seu usuário cerca de 1000 funções. Além disso, permite a criação e manipulação de arquivos flash e PDF e uma análise precisa de strings extremamente complexas, bem como a comunicação com diversos tipos de protocolos, como o IMAP, o POP3 e o DNS. Diante de tantos recursos, podemos afirmar com toda certeza que o PHP oferece ao desenvolvedor um enorme leque de possibilidades, que inclui, além dos já citados, o suporte à diversos tipos de banco de dados e à mecanismos como o *string-parsing* [14].

Por último, devemos destacar o seu Preço: o PHP é disponibilizado e distribuído de maneira inteiramente gratuíta, livre de quaisquer restrições sobre seu uso e redistribuição, o que torna a linguagem extremamente mais acessível aos desenvolvedores que queiram utilizá-la quando comparada à muitas outras linguagens similares [14].

2.1.6 Introdução à Padrões de Projeto: Arquitetura MVC

Quando pensamos na definição de padrão, pensamos em algo que pode ser utilizado como modelo para a elaboração de outra coisa. Segundo Gamma *et al.* (2000, p. 19, apud Alexander, 1977), "um padrão descreve um problema no nosso ambiente o cerne de sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira". Ainda que tal citação tenha sido elaborada inicialmente para descrever padrões no âmbito da construção civíl, ela também pode ser considerada completamente válida para o cenário do desenvolvimento de softwares [16].

Ainda de acordo com Gamma *et al.* (2000, p.19), no contexto do desenvolvimento de softwares, um padrão pode ser descrito através de quatro elementos básicos: **o seu nome**, que deve sintetizar em poucas palavras o padrão em si, o problema que ele resolve e as soluções por ele propostas; **o problema**, que é, em suma, a situação ou contexto em que o padrão em questão pode ser aplicado; **a solução**, que reúne as técnicas, mecanismos e procedimentos desenvolvidos pelo padrão para solucionar os problemas descritos na etapa anterior e, por último, **as consequências**, isto é, os resultados da aplicação do padrão, acompanhados de uma análise detalhada das vantagens e desvantagens de tal aplicação no contexto em questão [16].

Neste sentido, a abordagem Modelo/Visão/Controlador (*Model-View-Controlle*r/MVC) surge como uma das mais importantes arquiteturas no contexto de projetos de desenvolvimento de software, sendo utilizada como base para a construção e aplicação de inúmeros padrões de projeto utilizados por uma infinidade de desenvolvedores nos dias atuais. Ao dividir um projeto de software orientado a objetos em uma tríade de classes – Modelo, Visão e Controlador – e construir relacionamentos extremamente eficientes entre elas, a arquitetura MVC de desenvolvimento de software torna a implementação e a refatoração do projeto muito mais simples e prática, além de prover uma maior segurança ao software em questão, isolando as regras de negócios (sua lógica e implementação) da apresentação (interface gráfica vista pelo usuário) [14][16].

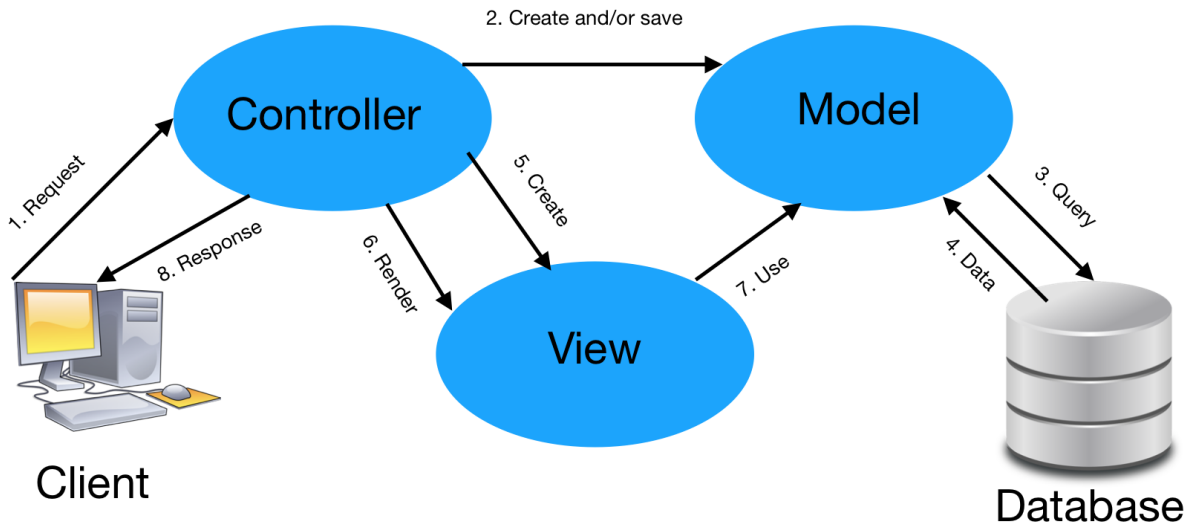
2.1.7 Componentes e Funcionamento do MVC

Como seu próprio nome já diz, a arquitetura MVC (*Model-View-Controller*) de desenvolvimento de software divide a aplicação a ser desenvolvida em três elementos ou tipos de

classe distintos: o modelo (model), a visão (view) e o controlador (controller). Cada um destes elementos ou tipos de classe é responsável por desempenhar uma função específica dentro da aplicação, ao mesmo tempo que deve estar interligada aos outros tipos de classe através de relacionamentos que são vitais para o funcionamento do software [14][16].

As classes **modelo** representam o objeto a ser trabalhado na aplicação, definindo as regras de negócio e especificando as características e comportamentos das entidades à serem representadas pela aplicação desenvolvida, enquanto as classes de **visão** são responsáveis pela formatação de tais dados e pela interface exibida ao usuário quando este utiliza a aplicação, em suma, é responsável pelo que chamamos de apresentação. Já as classes do tipo **controlador** são aquelas que coordenam as respostas da aplicação, definindo, de maneira geral, como a interface gráfica exibida ao usuário responde às entradas realizadas por este [14][16].

Figura 12. Diagrama de Funcionamento do MVC [17]



O diagrama acima ilustra o funcionamento de uma aplicação que utiliza a arquitetura MVC. Na primeira etapa, vemos o cliente da aplicação fazendo uma requisição – que, neste caso, pode ser o carregamento de uma página ou a realização de algum tipo de entrada, por exemplo). O recebimento e processamento de tal requisição é feito pelo controlador, que, no primeiro caso, irá criar e ordenar que a aplicação efetue o *render*, isto é, carregue a visão da página em questão. No segundo caso, a entrada de dados efetua a criação ou alteração de um modelo – tal alteração/criação é comunicada ao banco de dados através de uma query, e a resposta do banco é enviada de volta para o modelo. Ao fim de todas as etapas, a resposta da ação é enviada à visão e exibida ao cliente que efetuou a requisição em primeiro momento. De maneira geral, podemos dizer que o controlador usa o modelo e a visão, enquanto a visão utiliza apenas o modelo. O modelo, por sua vez, não faz uso de nenhum dos outros dois, sendo apenas utilizado por estes [14][16][17].

2.2 Etapas para o Desenvolvimento da Pesquisa

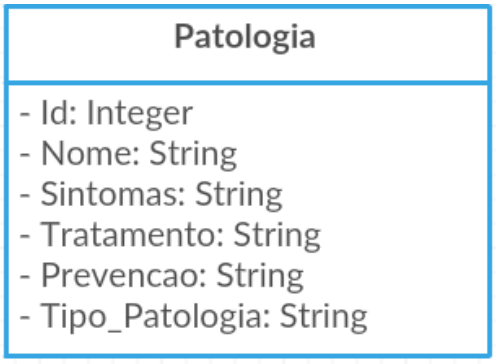
2.2.1 PHP Orientado a Objetos no Projeto Gerações

Como primeira etapa do desenvolvimento da pesquisa, esta seção abordará a aplicação dos conceitos de Orientação a Objetos abordados anteriormente no tópico 2.1.4 no âmbito do Projeto Gerações, desenvolvido no ano de 2019 na disciplina de Projeto e Desenvolvimento de Sistemas pelos alunos do 4º Ano do Curso Técnico Integrado em Informática do Campus São João da Boa Vista do Instituto Federal de São Paulo. Tal demonstração será realizada através do PHP, linguagem de programação utilizada no desenvolvimento do projeto, cuja história e conceitos básicos foram abordados no tópico 2.1.5 deste documento.

2.2.1.1 Classes em PHP no Projeto Gerações

Para demonstrar a modelagem e utilização de classes em PHP no Projeto Gerações, utilizaremos como exemplo a classe *Patologia*, utilizada pelos módulos 02 e 05 do projeto, representada pelo diagrama UML abaixo:

Figura 13. Representação UML da Classe Patologia



Como podemos observar, a classe *Patologia* contém 6 atributos, sendo um deles do tipo inteiro, e os outros cinco do tipo *String*. Além disso, outra observação importante é que todos eles possuem visualização definida como privada, e, portanto, só podem ser acessados e utilizados por esta classe e suas instâncias.

Ao construírmos a classe acima descrita utilizando a linguagem de programação PHP, obtemos o seguinte código:

Figura 14. Codificação PHP da Classe Patologia

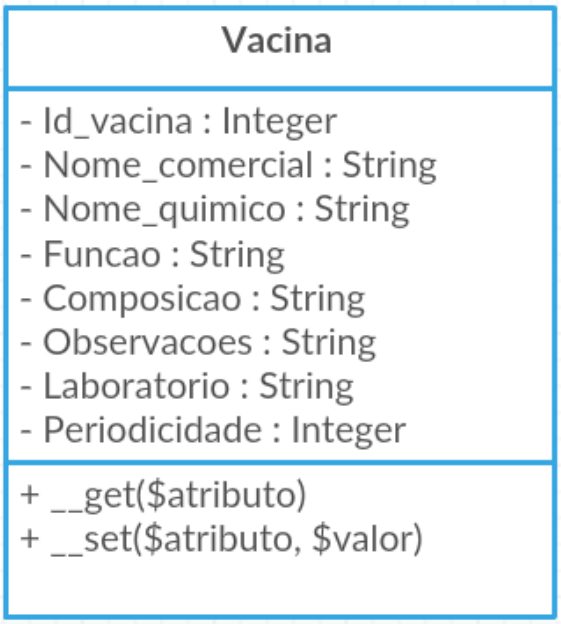
```php
class Patologia {

    private $id;
    private $nome;
    private $sintomas;
    private $tratamento;
    private $prevencao;
    private $tipo_patologia;
```

Podemos destacar como principais pontos a serem observados no código PHP acima a utilização da palavra reserva *class* para a declaração da classe, seguida de seu nome, neste caso, Patologia, além da ausência da declaração dos tipos de variáveis, que por sua vez estavam presentes no diagrama UML. Conforme vimos anteriormente, tal característica da linguagem nos possibilita uma grande flexibilidade e capacidade de adaptação ao código.

2.2.1.2 Encapsulamento em PHP no Projeto Gerações

Como vimos anteriormente, o Encapsulamento é um dos mecanismos mais importantes no âmbito da Programação Orientada a Objetos, sendo considerado um de seus quatro principais pilares. Para aplicar o mecanismo de Encapsulamento na aplicação desenvolvida no Projeto Gerações utilizando a linguagem PHP, faremos uso de duas técnicas: a definição de visibilidade para métodos e atributos e a implementação de métodos acessores (métodos *Get* e *Set)*, que serão a interface através da qual outras partes da aplicação externas à classe *Vacina* poderão acessar e efetuar modificações em seus atributos. Utilizaremos como exemplo da aplicação de tais técnicas a classe *Vacina*, representada pelo Diagrama UML abaixo:

Figura 15. Representação UML da Classe Vacina



No diagrama, podemos observar, na parte inferior da classe, a definição dos já citados métodos acessores, *Get* e *Set*, sendo que, em cada um deles há a passagem de pelo menos um parâmetro – no método *Get*, o parâmetro passado é o atributo que desejamos acessar, enquanto que no método *Set* temos também o parâmetro valor, que recebe o valor que desejamos que aquele atributo passe a assumir. Além disso, também é possível observar a definição da visibilidade dos atributos e métodos da classe, através do símbolo contido antes de seus nomes. Neste sentido, temos a definição de todos os atributos como privados, enquanto que os métodos são definidos com a visibilidade pública.

Para compreender melhor o funcionamento dos mecanismos utilizados no Encapsulamento, precisamos efetuar uma análise do código que define a classe Vacina, definido e construído conforme podemos observar na imagem abaixo:

Figura 16. Codificação PHP da Classe Vacina

```php
class Vacina {

    private $id;
    private $nome_comercial;
    private $nome_quimico;
    private $funcao;
    private $composicao;
    private $observacao;
    private $laboratorio;
    private $vacina_periodicidade;

    public function __get($atributo) {
        return $this->$atributo;
    }

    public function __set($atributo, $valor) {
        $this->$atributo = $valor;
    }

}
```

Assim como no diagrama UML, podemos observar a aplicação da visibilidade através das palavras *private* e *public* contidas antes da definição dos atributos e métodos da classe. Abaixo da declaração dos atributos, vemos a definição e implementação dos métodos acessores. Neste caso, utilizamos os chamados "Métodos Mágicos" *__get()* e *__set()*, cujos nomes são reservados no PHP e cujas funcionalidades já são pré-definidas pela linguagem. A utilização de tais métodos nos permite implementar apenas um método acessor *Get* e *Set* por classe, sem que haja a necessidade de criar um método acessor para cada atributo, como ocorre em outras linguagens de programação orientadas a objeto. De maneira geral, o funcionamento dos Métodos Mágicos *__get()* e *__set()* na classe acima apresentada ocorre da seguinte forma:

- **Método __get():** Quando necessitamos acessar um atributo de um objeto da classe *Vacina*, invocamos seu método __get() e passamos o nome do atributo como parâmetro. O método efetua o retorno do valor do atributo.
- **Método __set():** Quando precisamos efetuar a alteração no valor de um atributo de um objeto da classe *Vacina*, invocamos seu método __set() e passamos o nome do atributo e o valor que desjamos atribuí-lo como parâmetros. O método efetua a alteração do valor do atributo.

2.2.1.3 Instanciando e Definindo Objetos em PHP no Projeto Gerações

Agora que já vimos como construir e utilizar Classes, bem como o funcionamento de mecanismos de Encapsulamento como os métodos acessores *Get* e *Set* através do PHP Orientado a Objetos, iremos demonstrar como instanciar e definir objetos na linguagem. Para tal, utilizaremos como exemplo a mesma classe utilizada na seção anterior, a classe *Vacina*.

Figura 17. Instanciando e Definindo um Objeto da Classe Vacina

```php
<?php

// Importando a classe para que possamos utilizá-la
include_once "App\Model\Vacina.php";

// Instanciando objeto
$vacina = new Vacina();

// Atribuindo valores aos atributos do objeto
$vacina->__set('id', 1);
$vacina->__set('nome_comercial', 'VACINA INFLUENZA TRIVALENTE');
$vacina->__set('nome_quimico', 'VACINA INFLUENZA TRIVALENTE');
$vacina->__set('composicao', 'CEPAS VIRAIS INATIVAS, PURIFICADAS
E CULTIVADAS DENTRO DE CÉLULAS DE EMBRIÃO DE GALINHA');
$vacina->__set('laboratorio', 'LABORATÓRIO ABC');
$vacina->__set('vacina_periodicidade', 'ANUAL');
$vacina->__set('observacoes', 'PODE CAUSAR DOR, EDEMA, ERITEMA,
ENDURAÇÃO, FEBRE, MAL-ESTAR E MIALGIA.');

?>
```

Como podemos observar, a primeira etapa a ser realizada para que possamos instanciar e definir nosso objeto da classe *Vacina*, é importar a classe. Para isso, utilizamos o comando *include_once*, e passamos o endereço do diretório do arquivo *.php* em que a classe está contida em nossa aplicação, neste caso, *App\Model\Vacina.php.*

Com a classe devidamente importada para o arquivo em que estamos trabalhando, já podemos instanciar um objeto. Para tal, criamos uma variável, neste caso, a variável *vacina*, e atribuímos a ela um novo objeto da classe *Vacina*. Utilizamos, para tal, o operador *new*, que executa um método construtor – neste caso, o padrão, que define o valor de todas as variáveis da classe como *null*.

A última etapa do processo é a definição de nosso objeto, isto é, quando efetuamos o seu "preenchimento", atribuindo valores para os seus atributos. Para que possamos realizar tal atribuição, utilizaremos o método acessor *Set*, cujo funcionamento foi abordado e descrito na seção anterior. Para utilizá-lo, como podemos observar no trecho de código acima, basta chamarmos o método *__set()* de nosso objeto da classe *Vacina*, passando como parâmetros o atributo e seu respectivo valor.

2.2.1.4 Herança, Abstração e Polimorfismo em PHP no Projeto Gerações

Para exemplificar a aplicação dos mecanismos de Herança e Abstração em PHP no Projeto Gerações, utilizaremos como exemplo o sistema de conexão e acesso ao banco de dados construído em nosso Projeto para a realização de inserções, alterações, exclusões e consultas à base de dados construída pelos DBA.

Primeiramente, a fim de demonstrar o funcionamento da herança em nosso projeto utilizando a linguagem PHP, analisaremos o relacionamento estabelecido entre a classe *Connection* – responsável pela conexão à base de dados – e a classe *DAO* (*Data Access Object*), que possui os métodos básicos necessários para que a aplicação possa executar instruções em tal base.

Figura 18. Código PHP da Classe *Connection*

```php
abstract class Connection {

    private $conn;
    private $dbname;
    private $host;
    private $user;
    private $pass;

    public function __construct() {
        $this->dbname = "geracoes";
        $this->host = "localhost";
        $this->user = "root";
        $this->pass = "";
        try{
            $this->conn = new \PDO(
                "mysql:dbname=".$this->dbname.";host=".$this->host.";charset=utf8",
                $this->user,
                $this->pass
            );
        } catch (\PDOException $ex) {
            //echo "Ocorreu erro: ".$ex->getMessage();
            echo "Erro na conexão com o banco de dados! Tente novamente mais tarde!!!";
            die();
        }

    }

    protected function getConn() {
        return $this->conn;
    }

}
```

Como podemos observar no código contido na imagem acima, a classe *Connection* efetua a conexão da aplicação ao banco de dados através dos atributos *conn*, que diz respeito à conexão em si, *dbname*, que recebe o nome da base de dados (neste caso, nossa base de dados se chama gerações), *host*, que deve conter o endereço de acesso ao servidor do banco de dados (que, em nosso cenário, é a própria máquina em que estamos trabalhando, ou seja, *localhost*), e, por último, o par *user* e *pass*, que devem receber, respectivamente, o usuário e senha corretos para efetuar a conexão.

Abaixo da declaração de atributos, podemos ver a definição de um método construtor, que primeiramente atribui os valores acima citados aos atributos da classe, e posteriormente realiza uma tentativa de conexão (*try*) ao atribuir uma nova conexão PDO para o atributo *conn*, utilizando, para tal, os outros atributos da classe. Em caso de falha na conexão, o programa executa os comandos de dentro do escopo *catch*, responsáveis por emitir uma mensagem de erro. Por último, temos um método acessor *getConn()* responsável por efetuar o retorno da conexão no momento em que é invocado em alguma parte da aplicação.

Como podemos perceber, o processo de conexão ao banco de dados efetuado pela aplicação tem uma implementação um pouco delicada e complexa. Sendo assim, seria inviável ter de repetir o processo acima citado em todas as classes que necessitam efetuar a conexão com o banco de dados na aplicação, uma vez que isso configuraria uma clara desvantagem – tanto do ponto de vista prático, quanto do ponto de vista de desempenho e eficiência da aplicação, causando um consumo de tempo, linhas de código e processamento muito maiores de maneira completamente desnecessária. Por esse motivo, a melhor solução para evitar tal cenário é a utilização do mecanismo de Herança.

Figura 19. Código PHP da Classe DAO

```php
<?php

    namespace App;

    use FW\DB\Connection;


    abstract class DAO extends Connection{

        public abstract function inserir($obj);
        public abstract function excluir($obj);
        public abstract function alterar($obj);
        public abstract function buscarPorId($obj);
        public abstract function listar();

    }
?>
```

No código PHP ilustrado acima, podemos perceber o relacionamento de herança pela presença do operador *extends* – que, de maneira literal, nos diz que a classe *DAO* herda todos os atributos e métodos contidos na classe *Connection*. Desta forma, não será necessário replicar o código de *Connection* toda vez que precisarmos utilizar uma de suas funcionalidades.

Outro ponto importante a ser observado na imagem, é que a classe *DAO* é definida como uma classe abstrata, como podemos perceber pela presença do operador *abstract*, bem como os seus métodos – *inserir, excluir, alterar, buscarPorId e listar*. Por esse motivo, a classe *DAO* não pode ser

instanciada e seus métodos também não podem ser invocados pela aplicação. Tal arranjo nos possibilita criar uma aplicação de grande eficiência e com código extremamente limpo, pois diferentes classes necessitam utilizar os métodos contidos na classe *DAO*, sendo que em cada uma delas a implementação de tais métodos deve ocorrer de maneira distinta (Polimorfismo). Desta forma, cada uma das classes de acesso ao banco – conhecidas como "classes DAO" – cria sua própria assinatura para os métodos contidos em *DAO*, podendo, além disso, incluir outros métodos e atributos para complementar sua funcionalidade.

Figura 20. Código PHP da Classe *VacinaDAO*

```php
class VacinaDAO extends DAO{

    public function alterar($vacina) {

        $uDAO = new UsuarioDAO();
        $uDAO->verify();

        $sql = "UPDATE vacinas SET VAC_NOME_COMERCIAL = :NOME_COMERCIAL, VAC_NOME_QUIMICO =
            :NOME_QUIMICO, VAC_OBSERVACOES = :OBSERVACOES, VAC_COMPOSICAO = :COMPOSICAO,
            FK_VACINA_PERIODICIDADE_VPE_ID = :VACINA_PERIODICIDADE, FK_LABORATORIOS_LAB_ID =
            :LABORATORIO WHERE VAC_ID = :ID";
        $stmt = $this->getConn()->prepare($sql);

        $id = $vacina->__get('id');
        $nome_comercial = $vacina->__get('nome_comercial');
        $nome_quimico = $vacina->__get('nome_quimico');
        $observacoes = $vacina->__get('observacoes');
        $composicao = $vacina->__get('composicao');
        $vacina_periodicidade = $vacina->__get('vacina_periodicidade');
        $laboratorio = $vacina->__get('laboratorio');

        $stmt->bindParam(":ID",$id);
        $stmt->bindParam(":NOME_COMERCIAL",$nome_comercial);
        $stmt->bindParam(":NOME_QUIMICO",$nome_quimico);
        $stmt->bindParam(":OBSERVACOES",$observacoes);
        $stmt->bindParam(":COMPOSICAO",$composicao);
        $stmt->bindParam(":VACINA_PERIODICIDADE",$vacina_periodicidade);
        $stmt->bindParam(":LABORATORIO",$laboratorio);

        $stmt->execute();
    }
```

Na imagem acima, podemos observar um trecho do código da classe de acesso ao banco *VacinaDAO*, que reúne as funcionalidades responsáveis por fazer a comunicação de objetos da classe *Vacina* com o banco de dados. Ao realizar uma breve análise do trecho destacado, podemos destacar novamente a presença do operador *extends* indicando que a classe *VacinaDAO* é descendente da classe *DAO*, herdando, portanto, seus métodos que, por serem abstratos, necessariamente precisam ser implementados nas classes-filha. Neste sentido, também é válido destacar que isto ocorre, como podemos também destacar através da implementação do método *alterar* na imagem acima, contendo o código necessário para executar uma modificação no banco de dados que, neste caso, configura a assinatura de método nesta classe em específico. Verifica-se, então, a partir de tal análise, a presença também do mecanismo de Abstração no Projeto Gerações, aplicado nas classes acima expostas.

2.2.2 Arquitetura MVC definida no Projeto Gerações (Estrutura)

Agora que já descrevemos a utilização de elementos de Programação Orientada a Objetos no projeto através da linguagem PHP, é necessário redirecionar o foco da pesquisa de volta para o tema central deste documento: A Utilização da Arquitetura MVC no desenvolvimento do Projeto Gerações. Neste sentido, é de suma importância que façamos uma exposição e breve análise da estrutura das classes componentes do MVC (*Model*, *View* e *Controller*), além das classes de acesso ao banco de dados no Projeto. Esta seção é destinada, portanto, para tal fim.

2.2.2.1 Model no Projeto Gerações

Como já foi abordado anteriormente, as classes *Model* (Modelo) definem as regras de negócio e estabelecem, como seu próprio nome diz, um modelo dos objetos reais que precisam ser representados no projeto. No caso do Projeto Gerações, as classes Modelo são construídas a partir das entidades do banco de dados modelado pelos DBAs em conjunto com os professores da disciplina de PDS. Tal abordagem para o desenvolvimento permite uma maior facilidade na concepção e implementação de tais classes, uma vez que facilita, em momentos posteriores, a comunicação dos objetos de tais classes com o banco de dados através das classes *DAO*, além de facilitar o trabalho dos desenvolvedores.

Como exemplo do processo acima citado, demonstraremos a concepção e implementação da classe modelo *Idosos*, à começar pela entidade do banco de dados de mesmo nome, representada pela codificação SQL abaixo:

Figura 21. Tabela SQL para a Entidade "Idosos"

```
CREATE TABLE IDOSOS (
    IDS_DATA_INGRESSO DATE NOT NULL,
    IDS_FOTO VARCHAR(255) NOT NULL,
    IDS_SEXO INTEGER NOT NULL,
    IDS_USA_FRALDA_OU_FORRINHO BOOLEAN NOT NULL,
    IDS_RG VARCHAR(9) NOT NULL,
    IDS_CPF VARCHAR(11) NOT NULL,
    IDS_NOME_COMPLETO VARCHAR(255) NOT NULL,
    IDS_PRONTUARIO INTEGER AUTO_INCREMENT PRIMARY KEY,
    IDS_ATIVO BOOLEAN NOT NULL,
    IDS_DATA_NASCIMENTO DATE NOT NULL,
    IDS_DATA_FALECIMENTO DATE NOT NULL,
    FK_RESPONSAVEIS_USU_ID INTEGER NOT NULL,
    FK_HIGIENES_HIG_ID INTEGER
);
```

Como é possível perceber no código, temos uma série de campos que, de acordo com a abordagem descrita anteriormente, devem ser “replicados” para a classe modelo correspondente. É partindo de tal cenário que elaboramos o modelo abaixo e todas as outras classes de tal tipo presentes na aplicação.

Figura 22. Classe Modelo de Idoso em PHP

```php
namespace App\Model;

class Idoso {

    private $prontuario;
    private $nome;
    private $rg;
    private $cpf;
    private $sexo;
    private $estado;
    private $data_nasc;
    private $data_ingresso;
    private $cod_resp;
    private $fralda_ou_forrinho;

    public function __get($atributo) {
        return $this->$atributo;
    }

    public function __set($atributo, $valor) {
        $this->$atributo = $valor;
    }

}
```

Analisando o código acima, verifica-se que de fato há correspondência entre os atributos contidos na classe e os campos da tabela Idosos do banco de dados (com excessão das chaves estrangeiras e do campo foto, que não estão presentes por questões de planejamento).

2.2.2.2 Viewer no Projeto

Também estão presentes no Projeto Gerações as entidades responsáveis pela apresentação gráfica da aplicação – as *Visões* ou *Views* – sendo que estas compoem a primeira etapa do desenvolvimento da maioria dos projetos e do Gerações em específico, ainda na etapa de definição do *Template* e Identidade Visual e no posterior processo de Prototipação.

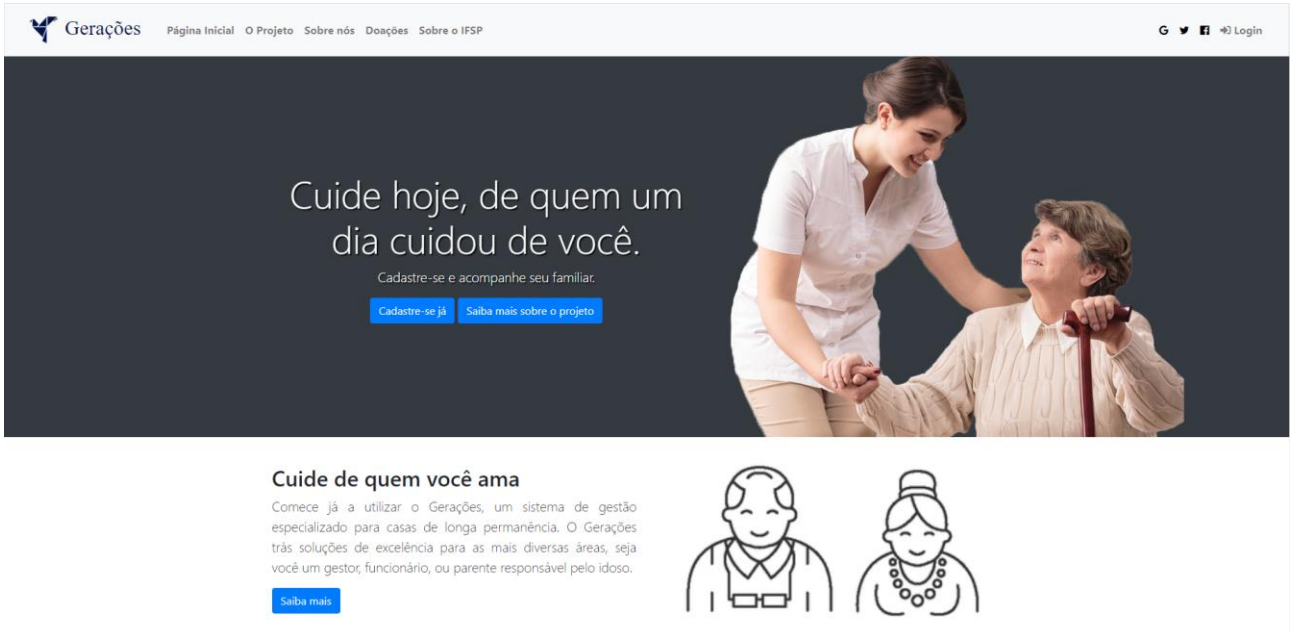
De maneira geral, as Visões do projeto são construídas através do *HTML5* unido aos estilos do *CSS* e do framework *Boostrap*, conforme podemos observar na imagem abaixo:

Figura 23. Código de uma View - Layout da Página Principal

```
<div id="nav-principal" class="collapse navbar-collapse">
    <ul class="navbar-nav">
        <li class="nav-item">
            <a href="/" class="nav-link font-weight-bold">Página Inicial</a>
        </li>
        <li class="nav-item">
            <a href="/sobre" class="nav-link font-weight-bold">O Projeto</a>
        </li>
        <li class="nav-item">
            <a href="/nos" class="nav-link font-weight-bold">Sobre nós</a>
        </li>
        <li class="nav-item">
            <a href="/doacoes" class="nav-link font-weight-bold">Doações</a>
        </li>
        <li class="nav-item">
            <a href="https://sbv.ifsp.edu.br" target="_blank"
            class="nav-link font-weight-bold">Sobre o IFSP</a>
        </li>
    </ul>
    <ul class="navbar-nav flex-row ml-md-auto d-none d-md-flex">
        <li class="nav-item p-2">
            <a href="" target="_blank" style="color:black;"> <i class="fa fa-google"> </i> </a>
        </li>
        <li class="nav-item p-2">
            <a href="" target="_blank" style="color:black;">
              <i class="fa fa-twitter"> </i></a>
        </li>
        <li class="nav-item p-2">
              <a href="" target="_blank" style="color:black;">
                <i class="fa fa-facebook-official"> </i> </a>
        </li>
        <li class="nav-item">
            <a href="/login" class="nav-link font-weight-bold">
              <i class="fas fa-sign-in-alt"></i> Login </a>
```

O código HTML associado à estilos do Bootstrap e CSS é responsável pela apresentação da Página Principal da aplicação, sendo que o resultado de sua implementação pode ser visualizado na imagem abaixo:

Figura 24. Representação Visual do Layout da Página Principal exibida pelo navegador



2.2.2.3 Controller no Projeto

No âmbito do Projeto Gerações, podemos classificar os Controladores em duas categorias distintas: os controladores de módulo – responsáveis por coordenar o carregamento das visões de cada módulo à partir da rota requirida pelo usuário – e os controladores das entidades – responsáveis por operações como cadastros, alterações e exclusões e que, portanto, estão sempre em contato com as classes de acesso ao banco de dados e modelos. De maneira geral, ambos os tipos de controladores desempenham, em conjunto, a função já estudada anteriormente – coordenar às respostas de ações desempenhadas pelo usuário nas visões.

Figura 25. Exemplo de Controlador de Módulo

```php
class Modulo01Controller extends Action{

    public function index(){
        $this->render('index', 'dashboard', '../');
    }

    public function feedback(){
        $this->render('feedback', 'dashboard', '../');
    }

    public function requisicoes(){
        $this->render('requisicoes', 'dashboard', '../');
    }

    public function gerenciar(){
        $this->render('gerenciar', 'dashboard', '../');
    }

    public function feedbacks(){
        $this->render('feedbacks', 'dashboard', '../');
    }

    public function respondeFeedback(){
        $this->render('respondeFeedback', 'dashboard', '../');
    }

    public function gerenciarConta(){
        $this->render('gerenciarConta', 'dashboard', '../');
    }

    public function minhaConta(){
        $this->render('minhaConta', 'dashboard', '../');
```

Como é possível observar no trecho de código acima, que representa o controlador do Módulo 01, possuímos uma série de métodos que possuem o nome de páginas da aplicação – eles são os métodos responsáveis pelo carregamento/*render* das visões do Projeto.

Figura 26. Exemplo de Controlador de Entidade/Objeto

```
class IdosoController extends Action{

    public function cadastrar(){
            $idoso = new Idoso();

            $idoso->__set('nome',$_POST['nome_idoso']);
            $idoso->__set('rg',$_POST['rg']);
            $idoso->__set('cpf',$_POST['cpf']);
            $idoso->__set('sexo',$_POST['sexo']);
            $idoso->__set('data_nasc',$_POST['data_nasc']);
            $idoso->__set('data_ingresso',$_POST['data_ingresso']);
            $idoso->__set('estado',$_POST['estado']);
            $idoso->__set('fralda_ou_forrinho',$_POST['fralda_ou_forrinho']);
            $idoso->__set('cod_resp',$_POST['cod_resp']);
            $idosoDAO = new IdosoDAO();

            if($idosoDAO->verificaCPF($_POST['cpf']))
            {
                ?>
                <script>
                    window.location.replace("https://disney.com.br");
                </script>
                <?php
                $idosoDAO->inserir($idoso);
            }else{
                $_SESSION['idosoError'] = "CPF";
            }

        header('Location: /md2/listar_cadastro');
        die();
    }
    public function alterar(){
        if(isset($_POST['prontuario']))
        {
            $AltIdoso = new AltIdoso();
            $AltIdoso->__set('prontuario',$_POST['prontuario']);
            $AltIdoso->__set('nome',$_POST['nome']);
```

Já o código contido na imagem acima exemplifica um controlador de entidade/objeto que executa, dentre outras ações, o cadastro, alteração e exclusão de dados inseridos pelo usuário em formulários contidos em páginas da aplicação. É possível notar, por esse motivo, que é realizada a instanciação de objetos da classe *IdosoDAO* para gravar tais operações na base de dados do Projeto.

Diante do exposto acima, é possível verificar, portanto, o funcionamento dos controladores como principais elementos de programação *back-end* da aplicação.

2.2.2.4 Classes de Acesso ao Banco no Projeto

O último elemento que necessita ser abordado nesta seção antes que possamos demonstrar o funcionamento do MVC na prática diz respeito às Classes de Acesso ao Banco de Dados, também conhecidas como *Classes DAO*. Tais classes são de extrema importância para o correto funcionamento da aplicação pois disponibilizam os meios através dos quais podemos efetuar o acesso à base de dados e nela realizar operações como inserções, exclusões, consultas e alterações.

Figura 27. Exemplo de Classe de Acesso ao Banco de Dados

```
class IdosoDAO extends DAO{

    public function alterar($idoso) {

        $uDAO = new UsuarioDAO();
        $uDAO->verify();

        $sql = "UPDATE idosos SET IDS_NOME_COMPLETO=:NOME, IDS_RG=:RG, IDS_CPF=:CPF,
        IDS_SEXO=:SEXO, IDS_DATA_NASCIMENTO=:DATA_NASC, IDS_DATA_INGRESSO=:DATA_INGRESSO,
        IDS_ATIVO=:ESTADO, IDS_USA_FRALDA_OU_FORRINHO=:FRALDA_OU_FORRINHO,
        FK_RESPONSAVEIS_USU_ID=:COD_RESP WHERE IDS_PRONTUARIO=:PRONTUARIO";
        $stmt = $this->getConn()->prepare($sql);

        echo "";

        $prontuario = $idoso->__get('prontuario');
        $nome_idoso = $idoso->__get('nome');
        $rg = $idoso->__get('rg');
        $cpf = $idoso->__get('cpf');
        $sexo = $idoso->__get('sexo');
        $estado = $idoso->__get('estado');
        $data_nasc = $idoso->__get('data_nasc');
        $data_ingresso = $idoso->__get('data_ingresso');
        $cod_resp = $idoso->__get('cod_resp');
        $fralda_ou_forrinho = $idoso->__get('fralda_ou_forrinho');

        $stmt->bindParam(":PRONTUARIO", $prontuario);
        $stmt->bindParam(":NOME",$nome_idoso);
        $stmt->bindParam(":RG",$rg);
        $stmt->bindParam(":CPF",$cpf);
        $stmt->bindParam(":SEXO",$sexo);
        $stmt->bindParam(":ESTADO",$estado);
        $stmt->bindParam(":DATA_NASC",$data_nasc);
        $stmt->bindParam(":DATA_INGRESSO",$data_ingresso);
        $stmt->bindParam(":COD_RESP",$cod_resp);
        $stmt->bindParam(":FRALDA_OU_FORRINHO",$fralda_ou_forrinho);

        $stmt->execute();
    }
```

Ao fazermos uma breve análise da imagem acima, que retrata um pequeno trecho do código da classe *IdosoDAO*, podemos observar a implementação de elementos para efetuar a comunicação com o banco de dados e executar nele uma instrução – neste caso, a de alteração dos dados de um registro específico: A varíavel *sql* recebe uma *query* que contem uma instrução de *Update*, a ser executada abaixo pela função *execute* após a construção dos parâmetros com atributos de um modelo.

2.2.3 Exemplo completo de MVC no Projeto Gerações

Nesta seção será apresentada a última etapa desta pesquisa, que consiste em realizar a demonstração da aplicação da Arquitetura MVC no Projeto Gerações através da exposição e análise de um exemplo completo de seu funcionamento, contemplando conceitos e elementos abordados anteriormente em todas as etapas deste documento – desde os conceitos de programação básica e Orientação a Objetos abordados no subcapítulo 2.1 e exemplificados anteriormente na seção 2.2.1, à estrutura das componentes básicas do MVC e sua implementação na aplicação desenvolvida no projeto exposta na seção anterior. Para tal, executaremos um cenário – o cadastro de um Medicamento na aplicação – apresentando o passo a passo do ponto de vista de um usuário que executa tal funcionalidade e descrevendo como a aplicação reage às ações por ele desempenhadas – o que, em suma, representa aquilo que desejamos investigar: o que ocorre por detrás da aplicação, no *back-end*, para que tais funcionalidades sejam executadas corretamente e respostas sejam exibidas ao usuário.

Inicialmente, partiremos do princípio, que, na visão do usuário, é a página inicial do painel administrativo do portal do Gerações, exibida após este efetuar corretamente o processo de autenticação (*login*) na aplicação. É a partir dela que o usuário pode acessar outras páginas do portal utilizando o menu lateral. Para que ele acesse a página de cadastro de medicamentos, é necessário clicar no submenu “Prescrições Médicas” e selecionar a opção “Medicamentos” após a abertura do *dropdown*. Uma vez na página de medicamentos, é possível observar uma tabela com todos os medicamentos já inseridos na base de dados da aplicação – nela, temos botões que nos possibilitam editar ou excluir os registros, encaminhando-nos para outras páginas. Porém, como nosso cenário descreverá o processo de cadastro, utilizaremos a opção “Inserir Medicamento” acima da tabela.

Figura 28. Página Inicial do Painel Administrativo do Gerações

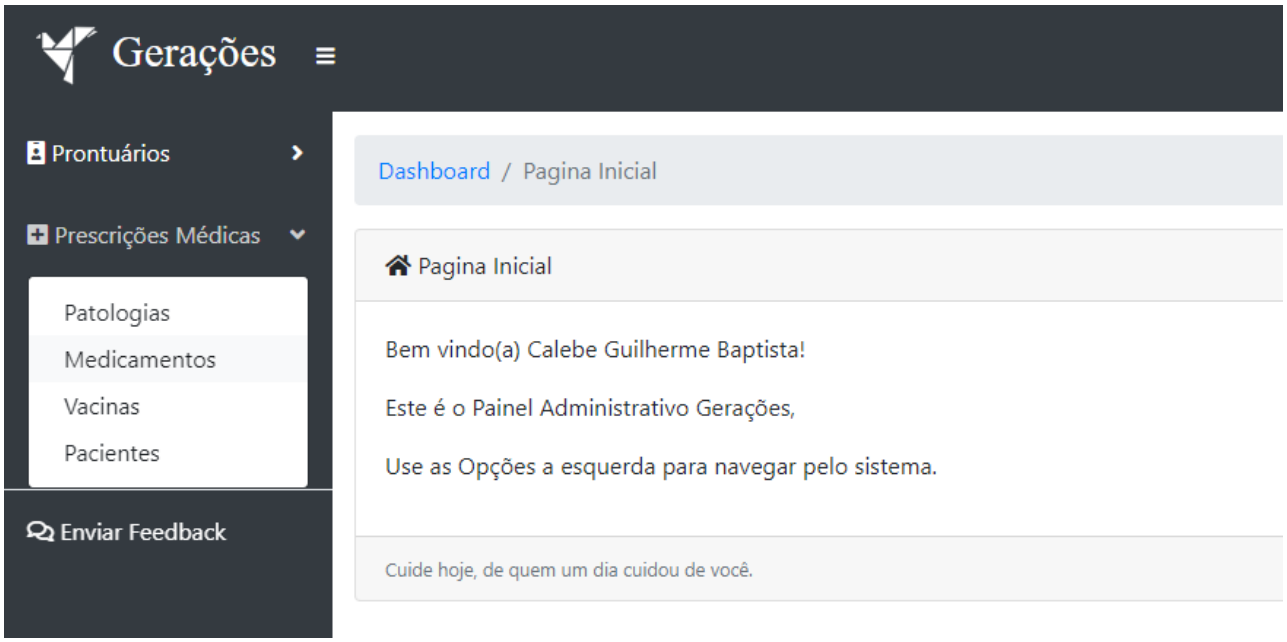


Figura 29. Página de Medicamentos da Aplicação

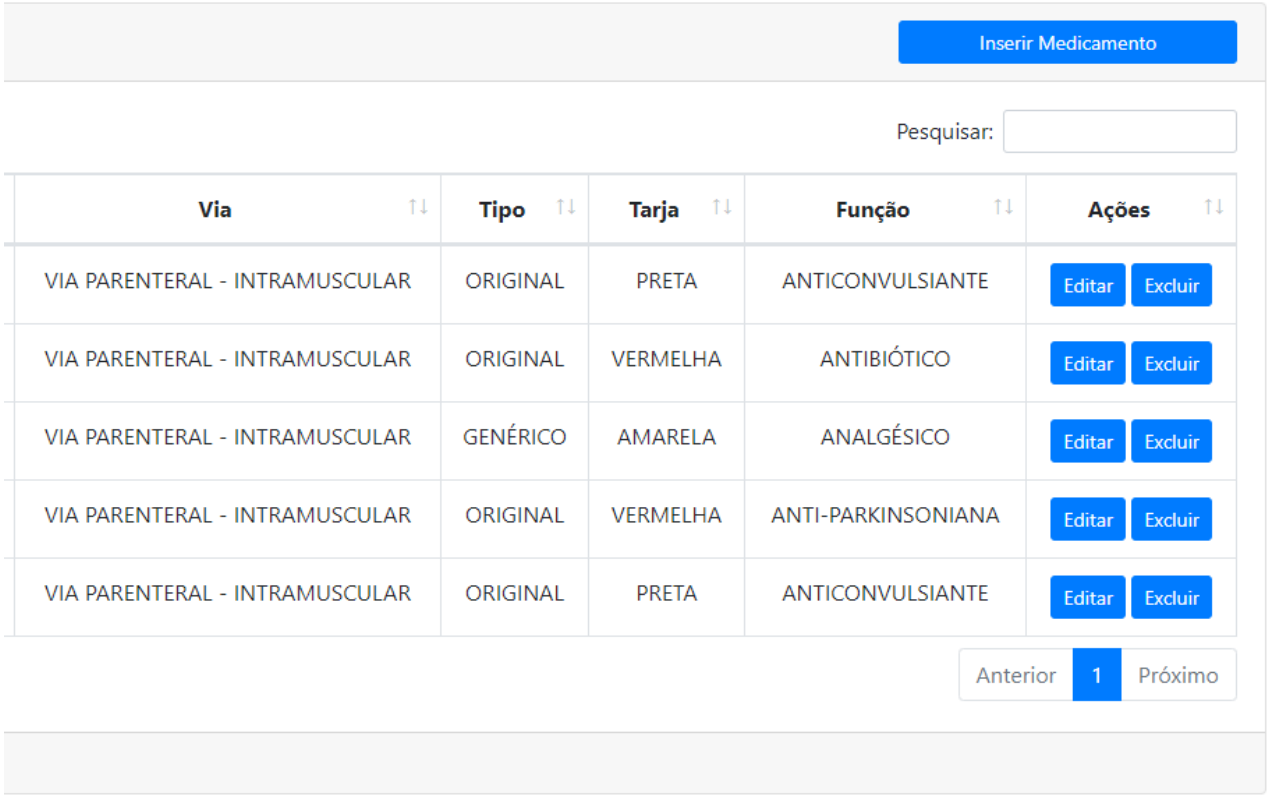


Figura 30. Formulário de Cadastro de Medicamentos

Até então, acessamos, através de apenas três cliques, três páginas distintas da aplicação que possuem como endereços, respectivamente: */md1/index*, */md5/medicamentos*, */md5/cadastrarMedicamento*. Para que a aplicação possa carregar as páginas corretamente, contudo,

é necessário que haja um processo de roteamento que conta com a atuação de diversas classes distintas: A classe *Action*, que define o método *render*, utilizado nos controladores de módulo para efetuar o carregamento das páginas, da classe *Route*, que define as rotas da aplicação e para onde (para qual controlador e qual método) cada uma delas aponta, e, por último, dos *Controladores* de módulo, que, conforme citado na seção anterior, são responsáveis pelo carregamento das visões requisitadas pelo usuário.

De maneira geral, o processo de carregamento de uma visão no Projeto ocorre da seguinte maneira: ao clicar em um link ou ser redirecionado para uma página, o usuário faz a requisição, através de seu navegador, de uma rota específica que aciona o método responsável pelo carregamento da visão no controlador de módulo correspondente. Tomando como exemplo a página de Cadastro de Medicamentos, ao clicarmos no botão "Inserir Medicamento" da página anterior, acessamos a rota */md5/cadastrarMedicamento*, que nos encaminha, através da classe *Route*, como podemos observar na imagem abaixo, para o método *cadastrarMedicamento()* contido no controlador do Módulo 05.

Figura 31. Definição das Rotas do Módulo 05 no Arquivo Route.php

```
//MOD05
new RouteUnique("md5-medicamentos",            "/md5/medicamentos",
    Modulo05Controller", "medicamentos"),
new RouteUnique("md5-cadastrarMedicamento",    "/md5/cadastrarMedicamento
    Modulo05Controller", "cadastrarMedicamento"),
new RouteUnique("md5-editarMedicamento",       "/md5/editarMedicamento",
    Modulo05Controller", "editarMedicamento"),
new RouteUnique("md5-excluirMedicamento",      "/md5/excluirMedicamento",
    Modulo05Controller", "excluirMedicamento"),
new RouteUnique("md5-vacinas",                 "/md5/vacinas",
    Modulo05Controller", "vacinas"),
new RouteUnique("md5-cadastrarVacina",         "/md5/cadastrarVacina",
    Modulo05Controller", "cadastrarVacina"),
new RouteUnique("md5-editarVacina",            "/md5/editarVacina",
    Modulo05Controller", "editarVacina"),
new RouteUnique("md5-excluirVacina",           "/md5/excluirVacina",
    Modulo05Controller", "excluirVacina"),
new RouteUnique("md5-patologias",              "/md5/patologias",
    Modulo05Controller", "patologias"),
new RouteUnique("md5-cadastrarPatologia",      "/md5/cadastrarPatologia",
    Modulo05Controller", "cadastrarPatologia"),
new RouteUnique("md5-editarPatologia",         "/md5/editarPatologia",
    Modulo05Controller", "editarPatologia"),
new RouteUnique("md5-excluirPatologia",        "/md5/excluirPatologia",
    Modulo05Controller", "excluirPatologia"),
new RouteUnique("md5-pesquisarIdoso",          "/md5/pesquisa",
    Modulo05Controller", "pesquisa"),
new RouteUnique("md5-analisesClinicas",        "/md5/analisesClinicas",
    Modulo05Controller", "analisesClinicas"),
```

Figura 32. Método de carregamento da página de Cadastro de Medicamentos

```
public function cadastrarMedicamento(){
    $labDAO = new LaboratorioDAO();
    $labs = $labDAO->listar();

    $viaDAO = new Via_MedicamentoDAO();
    $vias = $viaDAO->listar();

    $tipoDAO = new Tipo_MedicamentoDAO();
    $tipos = $tipoDAO->listar();

    $tarjaDAO = new Tarja_MedicamentoDAO();
    $tarjas = $tarjaDAO->listar();

    $funcaoDAO = new Funcao_MedicamentoDAO();
    $funcoes = $funcaoDAO->listar();

    $this->getView()->labs = $labs;
    $this->getView()->vias = $vias;
    $this->getView()->tipos = $tipos;
    $this->getView()->tarjas = $tarjas;
    $this->getView()->funcoes = $funcoes;
    $this->render('cadastrarMedicamento', 'dashboard', '../');
}
```

O método *cadastrarMedicamento()*, cujo código está presente na imagem acima, realiza, dentre outros, o salvamento dos resultados de listagens no banco de dados em variáveis que serão utilizadas na visão nos campos de seleção/*comboboxes*, mas, neste momento, direcionaremos o foco para o método responsável por efetuar o carregamento da página (*render)*, que é executado com a passagem de três parâmetros: O primeiro, *cadastrarMedicamento*, diz respeito ao nome do arquivo .php em que a visão que desejamos carregar está contida, enquanto o segundo indica em qual *layout* – isto é, qual template básico onde estão contidos os elementos presentes em todas as páginas, como barras de navegação e formatações gerais – a visão deve ser carregada (neste caso, o layout escolhido é o *dashboard*, ou seja, o do painel administrativo do Gerações). Por último, o terceiro parâmetro indica quantos diretórios devemos "voltar" para alcançar a pasta raiz do projeto e, portanto, poder acessar, dentre outros, os arquivos de estilo e *scripts* .js, que estão contidos na pasta *resources*. Como vimos, porém, o método *render()* é definido pela classe *Action*, sendo herdado pelos Controladores para que estes possam fazer sua execução. Para compreender melhor como tal método funciona, faremos uma análise deste, a partir do cenário abordado até este momento. Para isso, levaremos em conta a imagem abaixo, que representa a implementação da classe abstrata *Action*.

Figura 33. Implementação da Classe Action

```php
abstract class Action {

    private $view;

    function __construct() {
        $this->view = new \stdClass();
    }

    protected function getView() {
        return $this->view;
    }

    protected function render($view, $layout, $include=''){
        $this->view->page = $view;
        $this->view->include = $include;
        if(file_exists("App/View/$layout.php")){
            require_once "App/View/$layout.php";
        } else {
            $this->content();
        }
    }

    protected function content(){
        $classeAtual = get_class($this);
        $classeAtual = str_replace('App\\Controller\\', '', $classeAtual);
        $classeAtual = strtolower(str_replace('Controller', '', $classeAtual));

        require_once "App/View/$classeAtual/".$this->view->page.".php";
    }

}
```

De maneira geral, a classe *Action* utiliza os parâmetros passados anteriormente para fazer a requisição (*require_once)* do *layout* desejado, bem como definir qual *view* deve passar pelo mesmo processo de requisição, realizado pelo método *content()*, executado nos arquivos de *layout* (onde estão contidos apenas o cabeçalho e rodapé das páginas) para efetuar o carregamento da visão que preenche o "miolo" ou conteúdo de tais páginas.

Figura 34. Trecho de um Layout destacando a execução do método *content()*

```php
    <li class="nav-item dropdown active border-top">
      <a class="nav-link" href="/md1/feedback">
        <i class="far fa-comments"></i>
        <span>Enviar Feedback</span>
      </a>
    </li>
  </ul>

  <?php $this->content() ?>
```

Agora que compreendemos como se dá o carregamento das páginas no Projeto, podemos analisar como se dá o processo do Cadastro do Medicamento. Para tal, é necessário redirecionar o foco da análise de volta para a página do formulário, ilustrada pela Figura 30. Uma vez que o usuário preencha tal formulário e clique no botão para efetuar o cadastro, ele terá seus dados submetidos através do método *POST* e será redirecionado para a rota */md5/cadastroMedicamento* que, em nossa classe *Route*, ordena a execução do método *cadastrar()* do Controlador *MedicamentoController()*.

Figura 35. Classe *MedicamentoController* e implementação do método *cadastrar*

```
class MedicamentoController extends Action{

    public function cadastrar(){

        if (isset($_POST['nome_comercial']) && isset($_POST['nome_quimico'])
            && isset($_POST['laboratorio']) && isset($_POST['via'])
            && isset($_POST['tipo']) && isset($_POST['tarja'])
            && isset($_POST['funcao'])){

            $medicamentoDAO = new MedicamentoDAO();
            $medicamento = new Medicamento();
            $medicamento->__set('nome_comercial',$_POST['nome_comercial']);
            $medicamento->__set('nome_quimico',$_POST['nome_quimico']);
            $medicamento->__set('laboratorio',$_POST['laboratorio']);
            $medicamento->__set('via',$_POST['via']);
            $medicamento->__set('tipo',$_POST['tipo']);
            $medicamento->__set('tarja',$_POST['tarja']);
            $medicamento->__set('funcao',$_POST['funcao']);
            $medicamentoDAO->inserir($medicamento);
            header('Location: /md5/medicamentos');
            die();
        }
    }
```

Como podemos observar, o método *cadastrar()* é, como seu próprio nome diz, o responsável por efetuar o cadastro do Medicamento na aplicação e, para isso, ele necessita interagir com o Modelo de medicamento, cuja instância é atribuída à variável *medicamento*. Tal instância do modelo têm seus atributos preenchidos com os dados inseridos pelo usuário na página de cadastro, recebidos através da variável superglobal *$_POST*, que armazena as entradas realizadas no formulário de cadastro contido em tal *view*.

Após atribuir aos atributos do Modelo os seus respectivos valores utilizando o método *__set()*, o método *cadastrar()* ordena a execução de uma instrução de inserção na base de dados do Gerações. Porém, para que isso ocorra, é necessário que o controlador acesse a classe de acesso ao banco da entidade Medicamento – por esse motivo, podemos notar que ocorre uma instanciação da classe *MedicamentoDAO*, bem como a execução de seu método *inserir()* com o objeto contido no atributo *medicamento* sendo passado como parâmetro. Através da imagem abaixo é possível observar como tal inserção ocorre na classe *MedicamentoDAO*:

Figura 36. Método *inserir()* da Classe MedicamentoDAO

```
public function inserir($medicamento) {

    $uDAO = new UsuarioDAO();
    $uDAO->verify();

    $sql = "INSERT INTO medicamentos (MED_NOME_COMERCIAL,
    MED_NOME_QUIMICO, FK_VIA_MEDICAMENTOS_VIA_ID, FK_TIPOS_MEDICAMENTOS_TIP_ID,
    FK_TARJA_MEDICAMENTOS_TAR_ID, FK_FUNCAO_MEDICAMENTOS_FUN_ID,
    FK_LABORATORIOS_LAB_ID) VALUES (:NOMEC, :NOMEQ, :VIA, :TIPO,
    :TARJA, :FUNCAO, :LAB)";
    $stmt = $this->getConn()->prepare($sql);

    $nome_comercial = $medicamento->__get('nome_comercial');
    $nome_quimico = $medicamento->__get('nome_quimico');
    $via = $medicamento->__get('via');
    $tipo = $medicamento->__get('tipo');
    $tarja = $medicamento->__get('tarja');
    $funcao = $medicamento->__get('funcao');
    $laboratorio = $medicamento->__get('laboratorio');

    $stmt->bindParam(":NOMEC",$nome_comercial);
    $stmt->bindParam(":NOMEQ",$nome_quimico);
    $stmt->bindParam(":VIA",$via);
    $stmt->bindParam(":TIPO",$tipo);
    $stmt->bindParam(":TARJA",$tarja);
    $stmt->bindParam(":FUNCAO",$funcao);
    $stmt->bindParam(":LAB",$laboratorio);
    $stmt->execute();
}
```

No método acima, o objeto *medicamento* tem seus atributos armazenados em variáveis que são utilizados na construção dos parâmetros definidos na *query* SQL abaixo pelo método *bindParam.* Ao fim das construções de parâmetros, a instrução contida na *query (INSERT)* é executada pelo banco, concretizando a inserção dos dados inseridos pelo usuário no formulário da *view cadastrarMedicamento*.

Figura 37. Exibição dos Dados Inseridos pelo Usuário na Aplicação

Nome Comercial ↑↓	Nome Químico ↑↓	Laboratório ↑↓
ABCD	ABCD	IMUNIX - CLÍNICA DE IMUNIZAÇÃO

Desta forma, verifica-se, a partir da exposição e análise de um cenário real realizada nesta seção, a utilização e o funcionamento da arquitetura MVC no Projeto Gerações a partir da utilização de todos os conceitos pesquisados e expostos anteriormente, encerrando, portanto, o último objetivo específico deste trabalho.

3 Conclusões e Recomendações

Foi pensando no bem-estar da população idosa do município de São João da Boa Vista – considerado um dos melhores municípios do país para os indivíduos da "melhor idade" – que a ideia da elaboração de um projeto de software que atendesse às Casas de Longa Permanência, instituições que abrigam pessoas de mais de 60 anos, oferecendo-lhes cuidado completo (controle patológico, prescrições médicas, análises clínicas, alimentação, cuidados diários), além de atividades recreativas e de lazer. A partir de tal projeto, posteriormente nomeado como "Projeto Gerações", as Instituições de Longa Permanência – que, em sua maioria, não possuem qualquer tipo de sistema informatizado de controle – contariam com uma maior automatização de seus serviços, tornando-os melhores e mais eficientes, de maneira a oferecer uma maior qualidade de vida para aqueles que habitam as ILPs.

A partir do cenário inicial acima descrito e diante da necessidade vital da criação de um padrão de organização para o desenvolvimento que pudesse ser seguido por toda a equipe de desenvolvedores, estabeleceu-se como principal proposta para este trabalho a realização de um estudo acerca da concepção e funcionamento da arquitetura *Model-View-Controller* (MVC), com enfoque em sua aplicação e implementação no processo de desenvolvimento da aplicação *web* do já citado Projeto Gerações.

A primeira etapa a ser realizada na pesquisa era realizar uma apresentação de conceitos básicos relativos à Linguagens e Paradigmas de Programação, com foco principal nos conceitos de Programação Orientada a Objetos. Tal etapa foi concluída à partir de um Levantamento Bibliográfico realizado em obras de autores renomados do ramo da programação no capítulo 2.1 deste documento.

Já a segunda etapa consistia na apresentação do PHP – linguagem de programação voltada para o desenvolvimento *web*, e que fora utilizada durante todo o processo de desenvolvimento do Projeto Gerações. Tal apresentação fora realizada também durante a etapa de Levantamento Bibliográfico, em que descrevemos a história e as principais características que tornam tal linguagem extremamente eficiente para o desenvolvimento de projetos deste tipo.

Na etapa de número três, devíamos apresentar o conceito de Padrão de Projeto no âmbito do desenvolvimento de softwares, bem como definir a arquitetura MVC, foco principal deste trabalho. Isto foi feito também na seção 2.1, em que descrevemos os conceitos básicos que embasam Padrões de Projeto e MVC, bem como fizemos uma descrição completa das componentes essenciais do segundo, acompanhada de uma explanação completa acerca de como se dá o seu funcionamento em uma determinada aplicação.

Na etapa quatro, o objetivo principal era o de demonstrar a aplicação dos conceitos de POO abordados na etapa de número um no âmbito do Projeto Gerações. Para tal, realizamos uma exposição e análise de diversos trechos de código da aplicação que exemplificam a utilização de tais conceitos,

descrevendo como os mecanismos de Orientação a Objetos foram utilizados no Projeto e como estes foram extremamente importantes para o seu funcionamento.

Na etapa cinco, era necessário efetuar a especificação e realizar uma explanação acerca dos três tipos de classes principais que fazem parte de uma aplicação desenvolvida seguindo o padrão MVC. Para tal, exemplificamos, através da exposição e análise de trechos de código, onde cada uma destes três elementos básicos da arquitetura – Models, Views e Controllers – estavam presentes no desenvolvimento de nosso Projeto e de que maneira sua implementação colaborava para o correto funcionamento e eficiência deste.

Por último, na etapa de número seis, fez-se necessária a realização de uma apresentação, análise e explanação de como se deu a implementação e o funcionamento de tal padrão de desenvolvimento de softwares no Projeto Gerações. Para atingir tal objetivo, executamos um cenário real de utilização da aplicação – o cadastro de um medicamento por um usuário – e descrevemos, através de capturas de tela da aplicação e de trechos de seu código, como cada uma das etapas de tal processo ocorreu, do ponto de vista do usuário e do ponto de vista do *back-end* da aplicação, que processava e coordenava as respostas cabíveis para as requisições e entradas realizadas por este usuário.

Podemos concluir, portanto, a partir de toda a exposição realizada neste documento e das etapas acima descritas, que o objetivo principal deste trabalho – a realização de um estudo completo acerca da arquitetura MVC e sua implementação no Projeto Gerações – foi cumprido com sucesso, uma vez que, ao fim de todo o processo até aqui realizado, é possível compreender como tal arquitetura funciona e de que maneira se deu sua aplicação no processo de desenvolvimento da aplicação proposta pela disciplina de PDS no ano de 2019.

Como principal ponto positivo acerca da elaboração deste trabalho, é possível destacar, primeiramente, que a realização do Levantamento Bibliográfico é capaz de nos fazer relembrar inúmeros conceitos de suma importância no âmbito de desenvolvimento de softwares. Além disso, através da análise completa do funcionamento da aplicação, podemos pensar em pontos positivos e negativos para a implementação do MVC: Ao mesmo tempo que possibilita uma maior organização e segurança para projetos com a separação do usuário da lógica da aplicação, este acaba implicando em uma modelagem muito mais complexa, que pode acabar "engessando" e atrasando o processo de desenvolvimento, sobretudo em equipes que não estão familiarizadas com tal padrão. Como ponto negativo da elaboração do trabalho como um todo, é necessário destacar principalmente os problemas decorridos de uma má gestão do tempo.

Para a realização trabalhos futuros, é completamente válido destacar a necessidade da explanação acerca de outros Padrões de Projeto como os observados na obra de Erich Gamma.

4 Referências Bibliográficas

[1] Portal da Rede Federal de Educação Profissional, Científica e Tecnológica. **Histórico da Rede Federal de Educação Profissional, Científica e Tecnológica**. Disponível em: http://redefederal.mec.gov.br/historico. Acesso em: 23 de Agosto de 2019.

[2] Portal da Rede Federal de Educação Profissional, Científica e Tecnológica. **Expansão da Rede Federal de Educação Profissional, Científica e Tecnológica**. Disponível em: http://redefederal.mec.gov.br/historico. Acesso em: 23 de Agosto de 2019.

[3] Instituto Brasileiro de Geografia e Estatística. **Cidades e Estados – São João da Boa Vista.** Disponível em: https://www.ibge.gov.br/cidades-e-estados/sp/sao-joao-da-boa-vista.html. Acesso em: 05 de Setembro de 2019.

[4] Portal do Instituto Federal de Educação, Ciência e Tecnologia de São Paulo – Campus São João da Boa Vista. **O Instituto Federal de São Paulo e o Câmpus de São João da Boa Vista.** Disponível em: https://www.sbv.ifsp.edu.br/sobre-campus. Acesso em: 05 de Setembro de 2019.

[5] Portal do Instituto Federal de Educação, Ciência e Tecnologia de São Paulo – Campus São João da Boa Vista. **O Instituto Federal de São Paulo e o Câmpus de São João da Boa Vista.** Disponível em: https://www.sbv.ifsp.edu.br/index.php/component/content/article/64-ensino/cursos/168-tecnico-integrado-informatica. Acesso em: 05 de Setembro de 2019.

[6] Enciclopédia do Instituto Federal de Educação, Ciência e Tecnologia de São Paulo – Campus São João da Boa Vista. **Prática de Desenvolvimento de Sistemas (PDS)**. Disponível em: https://sbv.ifsp.edu.br/wiki/index.php/Pr%C3%A1tica_de_Desenvolvimento_de_Sistemas_(PDS)_(T%C3%A9cnico_Integrado_em_Inform%C3%A1tica). Acesso em: 05 de Setembro de 2019.

[7] G1 São Carlos e Araraquara. **Pesquisa aponta São João da Boa Vista como melhor cidade para idosos**. Disponível em: http://g1.globo.com/sp/sao-carlos-regiao/noticia/2017/03/pesquisa-aponta-sao-joao-da-boa-vista-como-melhor-cidade-para-idosos.html. Acesso em: 05 de Setembro de 2019.

[8] ROMANO, B. L. **Macrorequisitos dos Módulos do Projeto Mais Saúde São João**. 2018. Disponível em: https://sites.google.com/site/blromano/disciplinas/pds2014. Acesso em: 05 de Setembro de 2019.

[9] WILLRICH, Roberto. **Linguagens de Programação.** Disponível em: http://algol.dcc.ufla.br/~monserrat/icc/Introducao_linguagens.pdf. Acesso em: 25 de setembro de 2019.

[10] LEAL, Marcus. **Sintaxe e Semântica.** Disponível em: http://www.inf.puc-rio.br/~inf1621/sintaxe.pdf. Acesso em: 25 de setembro de 2019.

[11] SIQUEIRA, Fernando. **A Sintaxe de uma Linguagem de Programação.** Disponível em: https://sites.google.com/site/proffernandodesiqueira/disciplinas/paradigmas-de-linguagens-de-programacao/aula-2. Acesso em: 25 de setembro de 2019.

[12] GOULART, Cristian. **Paradigmas de Programação.** Disponível em: http://www.petry.pro.br/sistemas/programacao1/materiais/artigo_paradigmas_de_programacao.pdf. Acesso em: 25 de setembro de 2019.

[13] DALL'OGLIO, Pablo. **PHP: Programando com Orientação a Objetos.** 2ª Edição. São Paulo: Editora Novatec, 2009.

[14] GILMORE, Jason W. **Dominando PHP e MySQL.** 1ª Edição. Rio de Janeiro: Editora Alta Books, 2011.

[15] SINTES, Antony. **Aprenda Programação Orientada a Objetos em 21 dias.** 1ª Edição. São Paulo: Perason Education do Brasil, 2010.

[16] GAMMA, Erich. **Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos.** 1ª Edição. São Paulo: Bookman, 2000.

[17] RUGER, Torsten. **Model View Controller: Theory and Practice.** Disponível em: https://medium.com/rubydesign/model-view-controller-theory-and-practice-e5db514c18d9. Acesso em: 15 de outubro de 2019.