Trabalho Prático 01

Nome: João Pedro Silva Bicalho

Matrícula: 21.1.4004

1. Introdução ao trabalho

O trabalho consiste na implementação de um sistema de controle de funcionários. Foi requisitado que, dependendo da ocupação, o programa haja funcionalidades diferentes. Foram exigidas algumas condições em relação à algumas opções dos menus. O programa foi desenvolvido na linguagem C++, utilizado o VS Code como IDE. Foi utilizado o paradigma de programação orientada à objetos e a aplicação de seus pilares, assim como propõe a disciplina.

2. Programação Orientada a Objetos e sua aplicação

A aplicação do paradigma estudado na disciplina é fundamental no desenvolvimento do trabalho. A criação de classes (objetos) que se relacionam entre si, a criação de interações entre as classes, como amizade e herança, encapsulamento de classes, até transformação das classes foram extremamente aplicados no desenvolvimento do programa.

A utilização do encapsulamento foi recorrente durante todo o código. A ideia de isolar os atributos da classe, que são privados para evitar qualquer alteração desnecessária, é aplicada com a criação de funções *tipo getAtributo( )* e *void setAtributo(tipo)* que respectivamente recebem e alteram o valor do atributo de uma instância de uma classe.

Como boa prática para criação de instâncias de objetos, foi criado também construtores e destrutores. O primeiro tem como função atribuir um valor determinado, ou padrão, para a criação de uma nova instância de classe, de modo em que suas variáveis não armazenem lixo de memória. Já o segundo, tem a função, literalmente, de destruir a instância do objeto, de modo que o que foi alocado pelo objeto seja liberado.

Também, em relação as classes, foi utilizado alguns padrões de relacionamentos entre classes, como a Herança e classes/funções amigas. A herança foi aplicada em algumas classes para que elas recebam os atributos das outras, evitando a repetição de código e promovendo também, a aplicação de outro conceito: o polimorfismo. Já as classes amigas foram utilizadas para que uma classe consiga ter acesso aos atributos da hora, que foi o método utilizado em uma das condições apresentadas no trabalho.

Além disso, a aplicação dos conceitos de polimorfismo foi fundamental na estruturação do raciocínio lógico do programa. A ideia de salvar as coisas em uma classe base, fazer um casting na instância e conseguir acessar dados da classe derivada se deu necessária para poder estruturar com mais praticidade as “Pessoas” do programa

3. Diagrama UML

O diagrama foi desenvolvido no Wondershare EDrawMax,versão de avaliação. Ele consiste numa visualização mais concreta das relações entre as classes, de modo a simplificar o entendimento do código. *(Visualizar no arquivo .zip)*

4. Diretivas de compilação

Foi utilizado o próprio prompt do VSCode para compilar e executar o TP, no SO Windows 10 x64. As instruções para compilar e executar o código estarão dentro de um arquivo *.txt* no *.zip* do trabalho também. A seguir estão elas:

```
g++  modoFuncionario.cpp  modoGerente.cpp  chefe.cpp  funcionario.cpp  horario.cpp
menu.cpp pessoa.cpp vendas.cpp -c -Wall

g++ modoFuncionario.o modoGerente.o chefe.o funcionario.o horario.o menu.o pessoa.o
vendas.o  main.cpp -o e.exe -Wall

.\e.exe
```

(diretivas de compilação)

```
PS C:\Users\joaop\OneDrive\Área de Trabalho\POO\TrabalhoPratico> g++ modoFuncionario.cpp modoGerente.cpp chefe
.cpp funcionario.cpp horario.cpp menu.cpp pessoa.cpp vendas.cpp -c -Wall
```

```
PS C:\Users\joaop\OneDrive\Área de Trabalho\POO\TrabalhoPratico> g++ modoFuncionario.o modoGerente.o  chefe.o
funcionario.o horario.o menu.o pessoa.o vendas.o  main.cpp -o e.exe -Wall
```

```
PS C:\Users\joaop\OneDrive\Área de Trabalho\POO\TrabalhoPratico> .\e.exe
Entrar como:
1.Gerente
2.Funcionario
3.Sair
```

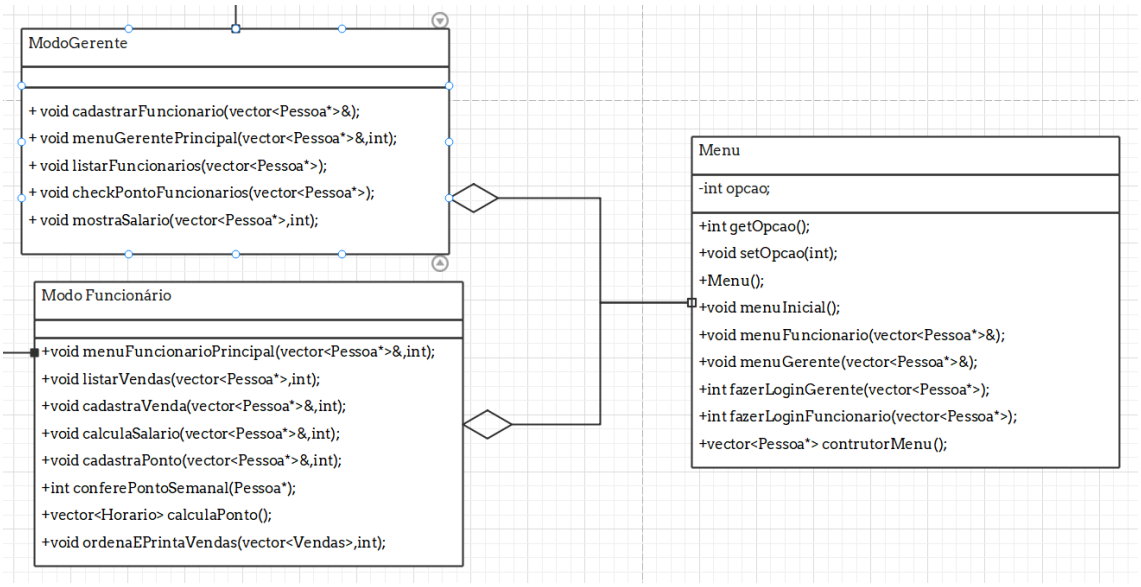
(aplicação das diretivas)

5. Explicação do código

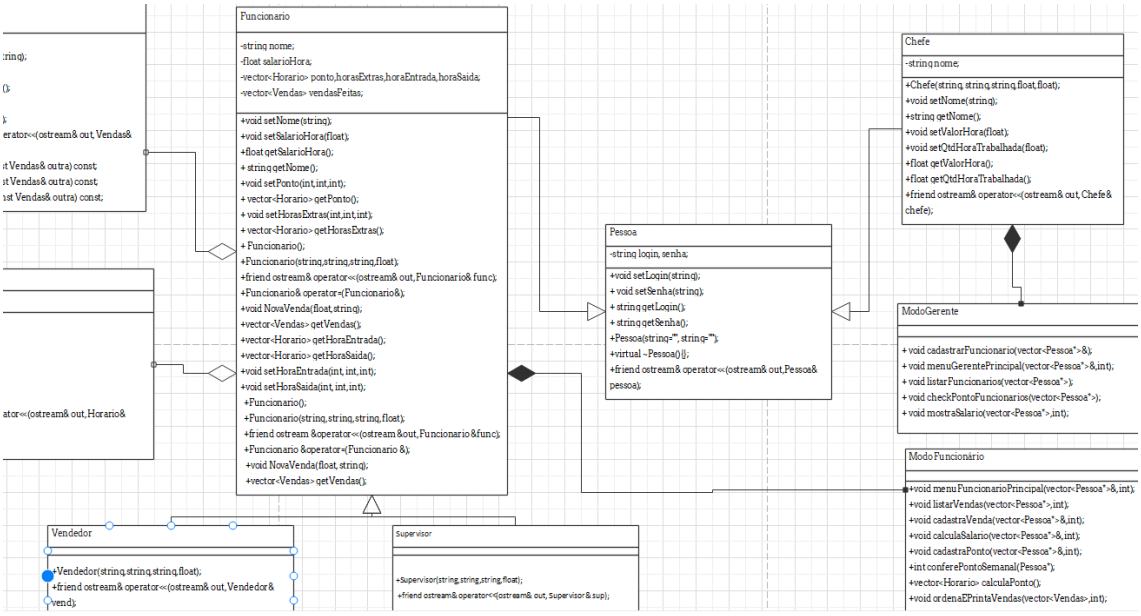
a. Objetos/Classes

As classes, como visto no diagrama UML, são divididas em 3 funcionalidades: Os menus, que são responsáveis por chamar funções e manipular os dados recebidos e existentes, as Pessoas, que vão armazenar dados específicos a sua função na empresa e manipulá-los, e as de tipo de armazenamento, que vão servir para organizar melhor os dados de cada pessoa e facilitar a alteração delas de acordo com a intenção do usuário.

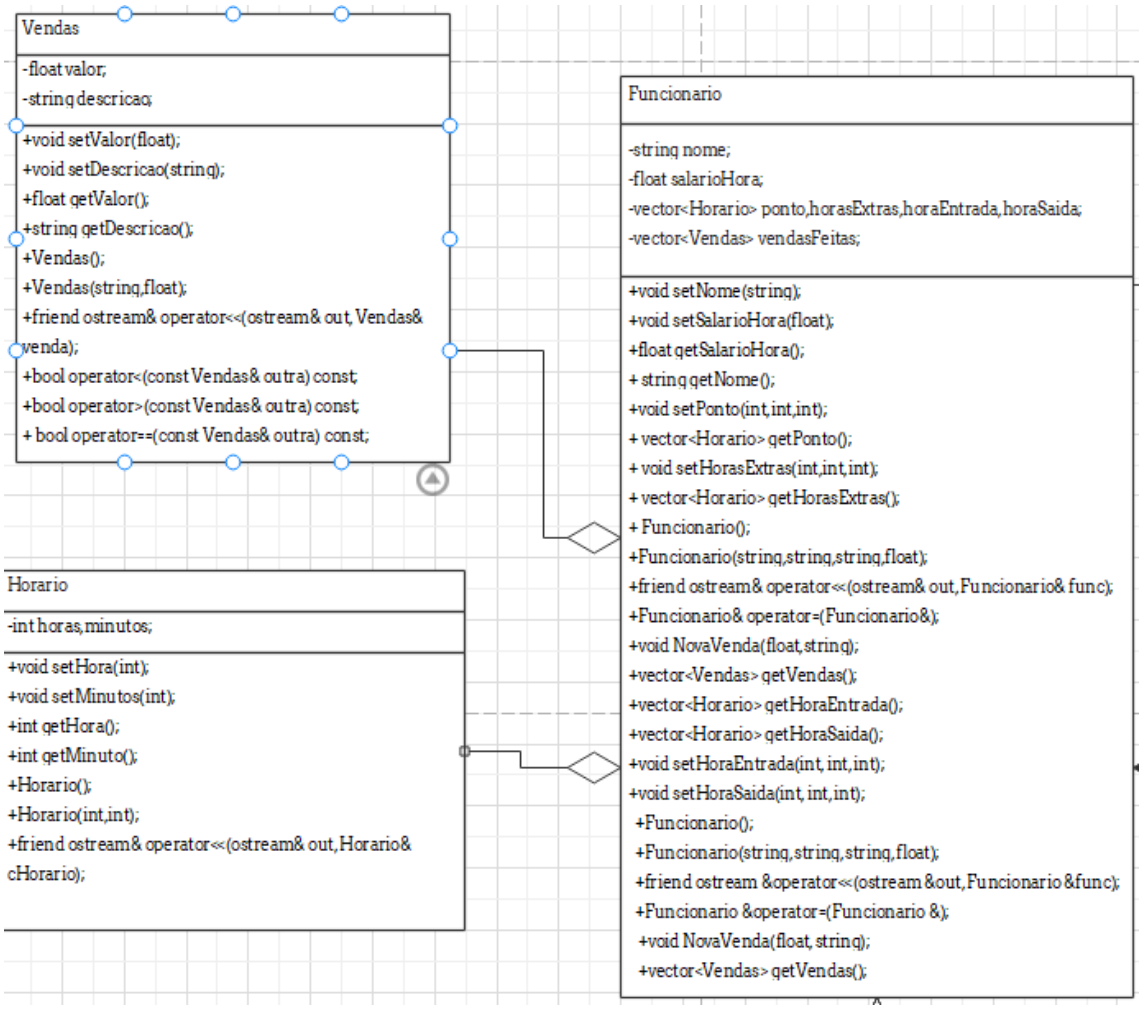
Os objetos menus são baseadas em: *Menu, ModoGerente e ModoFuncionario.* São elas quem fazem a função de ligar as intenções do usuário com as funcionalidades do programa. Seja logar no modo Funcionário ou no modo Chefe, receber requisições de cálculo de salário, cálculo e cadastro de ponto, cadastro de vendas, entre outros.



Já as pessoas, descritas como as classes *Pessoa, Funcionario, Vendedor, Supervisor, Chefe* vão armazenar as informações de cada pessoa que irá ser adicionada ao programa, e com base na hierarquia de classes, os atributos vão tornando-se mais específicos.

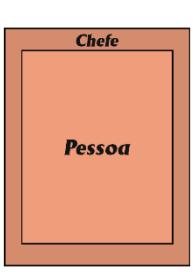


E por último, vem *Vendas* e *Horario* que vão ser importantes para organização do cadastro de vendas e para a manipulação de horários de pontos e horas extras respectivamente.



b. Relação entre Classes

Foi adicionada a herança entre as classes *Pessoa* -> *Funcionario* -> *(Supervisor/Vendedor)* e *Pessoa* -> *Chefe* para que isso possibilite a criação de um vetor **polimórfico** do tipo *Pessoa* que armazena todas as pessoas do programa. Outra relação foi a de ***friend*** entre *Vendedor* e *Supervisor* para que seja possível o *Supervisor* tenha acesso às vendas dos *Vendedores*, sendo assim ser possível calcular o salário do modo que foi condicionado nas especificações do programa.



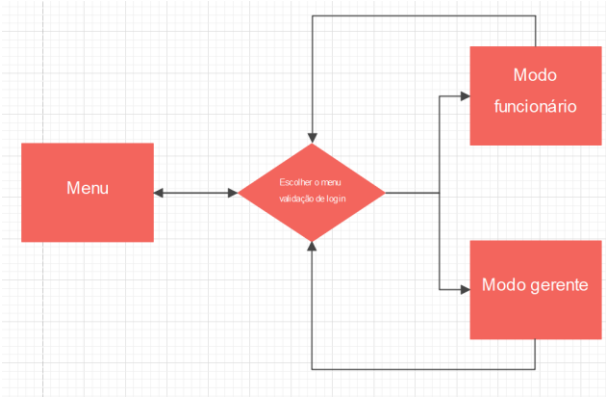
As imagens acima são uma simulação do vetor polimórfico: ele pode carregar tanto funcionários quanto chefes, já que eles têm uma parte de *Pessoa* dentro deles. Sendo que quanto mais escuro, mais há informações específicas em cada instância.

c. Ideia central

O pilar do código inteiro é o *vector<Pessoa*>* criado logo no início do código com o cadastro interno de um chefe. Por ser um ponteiro de *Pessoa*, o vetor, por meio de casting/polimorfismo, consegue armazenar o endereço de memória de classes derivadas. Assim, sempre que desejar manipular, seja criar uma pessoa ou alterar e receber informações, utiliza-se de uma conversão de tipo, para fazer a manipulação e depois outra conversão para salvar no vetor. Assim, de acordo com a transição de menus durante a segmentação do programa, é passado esse vetor por referência, para que as alterações sejam feitas no local da variável original, mesmo que o "modo" do programa seja alterado.

d. Explicação mais afunda do código

Em síntese, o código consiste em 3 menus, o principal e os menus de chefe e funcionário, que são independentes. O principal é interligado nos outros por meio de uma validação de login, em que usuário precisa ter credenciais adequadas para acessar o outro menu. O diagrama a seguir exemplifica bem a relação de acesso e o deslocamento entre os menus:



Já no início do menu, é chamada uma função *construtorMenu()* que tem como objetivo criar um Chefe para que a partir dali ele crie outros *funcionários*.Com essa criação, será possível também acessar o modo funcionário, que terá uma lógica de autenticação. Vale ressaltar que a função citada permite a criação de várias pessoas, baseando-se apenas no acréscimo de código, de modo a facilitar a fase de testes.

Portanto, sabendo da validação, não tem como utilizar as credenciais para acessar o menu que não foi designado pois, por meio de uma conferência de *casting* analisa cada pessoa presente no *vector<Pessoa*>* e só conseguirá acesso caso a ocupação da pessoa confira com o modo a ser logado. Vale ressaltar também que a funcionalidade do login tem o retorno da posição de qual login e senha foram validados, de modo que quando entrar no modo desejado, será já de conhecimento das funções a posição em que a *Pessoa* será manipulada. No código abaixo, fica o exemplo do código do login de um *chefe/gerente*

```
1       for(vector<Pessoa*>::size_type i=0;i<p.size();i++){
2           //estrutura de login credenciada:
3           //SO CONSEGUE ENTRAR NO MENU GERENTE QUEM EH GERENTE
4           Pessoa* alguem = p[i];
5           [[maybe_unused]] Funcionario* func;
6           if (( func = dynamic_cast<Funcionario*>(alguem))) continue;
7           //Se no vetor for funcionario, o for(l1) continua e ñ confere
8           if(p[i]->getLogin() == user && p[i]->getSenha() == password){
9                cout<<"\n\n****LOGIN VALIDADO!!!****\n";
10               tentativa = ACERTO;//enum de acerto ou erro para esclarecer codigo
11               posicao = i;
12               return i;
```

Assim, a partir da validação, existe uma conferência da posição, de modo que se for -1 o valor de retorno da função de login, simboliza que houve uma falha na validação. Assim, volta para função inicial e entra no loop na opção dos menus ou sair. Caso o contrário, se cria uma instância de um menu do modo validado, em que liberará as respectivas funcionalidades.

```
void Menu::menuGerente(vector<Pessoa*>& p){

    int pos=fazerLoginGerente(p);// retorna posicao de login der certo ou -1 se falhar
    if (pos==-1) return;
    ModoGerente novoMenu;
    novoMenu.menuGerentePrincipal(p,pos);
}
```

Exemplo do menu do chefe

Adentrando ao modo gerente, se abre um menu com as opções de o que manipular que foram dadas no trabalho.

A primeira, cadastro de funcionário, vai receber do usuário os dados necessários e vai criar utilizando um construtor por parâmetros e adicionado no vetor pilar *vector<Pessoa*>* por meio do *push_back()*, que foi passado por referência na função. É de importância saber que há uma função que valida a criação de usuários de modo que os logins e senhas não podem ser iguais aos já criados.

Depois, vem a listagem dos funcionários que basicamente imprime por meio de uma sobrecarga do operador <<, os funcionários do vetor que é recebido de parâmetro.

Já na terceira função, lista a hora de entrada, saída, horas trabalhadas e as horas extras de acordo com o que foi condicionado na proposta do trabalho. **Vale ressaltar que foi considerado o mês como 4 semanas de 7 dias, totalizando 28 dias.** Assim, por meio de iterações se imprime o mês. É importante lembrar também que há a verificação dos horários da jornada de trabalho tanto diária quanto semanal, para obedecer às 8 e 10 horas respectivamente.

Já a última se trata apenas de um cálculo de salário relacionando as horas trabalhadas e o valor de sua hora.

Partindo para o modo dos funcionários, vale ressaltar inicialmente as classes *Horário* e *Venda*, que são necessárias para a organização das manipulações de dados dos funcionários. Uma das funcionalidades, a criação de pontos, recebe as horas de entrada e saída de um dia especificado, confere a entrada de dados e calcula o horário trabalho e as horas extras. Depois disso, ele define na posição *[dia-1]* de *ponto, horasExtras, horaEntrada e horaSaida* os horários para ficar armazenados e permitir também calcular as horas semanais trabalhadas.

Uma boa observação a se fazer é sobre a posição *dia-1*. Sabendo que a contagem de posições de um vetor começa em 0 e os dias de um mês começam em 1, a relação dias e posição do vetor fica:

```
posicaoVetor = dia – 1

ex. dia 20 => 20-1 = posicaoVetor => 19 = posicaoVetor

a posicao que vai armazenar os dados do dia 20 é a 19.
```

A segunda funcionalidade está muito atrelada com a terceira. A criação de vendas e a listagem são operações existentes no programa. É possível criar com base no input de dados, a descrição e o valor da venda, e esta por consequência é adicionada a um *vector<Vendas>* que fica em cada instância dos objetos tipo *Funcionario*. Porém a listagem das vendas funciona diferente de acordo com o tipo de funcionário que você é: se for *Vendedor* lista as próprias vendas, caso seja *Supervisor* lista as de todos os funcionários. Isso é feito por meio de uma tentativa de cast:

```
if((sup =  dynamic_cast<Supervisor*>(func))){

    for(vector<Pessoa*>::size_type i=0;i<pessoas.size();i++){
        alguem = pessoas[i];
        [[maybe_unused]] Funcionario* func;
        if((func = dynamic_cast<Funcionario*>(alguem))){
```

É necessário pontuar que essa função de listar recebe o *vector<Pessoa*>* por cópia, uma vez que só acontece a extração de dados,ou seja, não há a alteração das informações no *vector.*

A última opção do menu se trata do cálculo do salário. Este método também é alterado o jeito que funciona de acordo com a ocupação: as horas extras têm um valor maior se for *Supervisor*, do que quando for *Vendedor,* além do valor percentual de vendas.

Vale ressaltar que para a criação de novas pessoas, o programa utiliza a palavra-chave *new* que cria dinamicamente uma pessoa no cargo adequado por meio de um vetor de pessoa (faz o *casting* automático). Sempre que há essa criação, a pessoa é adicionada ao vetor base *vector<Pessoa*>*. Assim, antes do programa ser finalizado ele irá deletar essas instâncias dinâmicas, por meio de um *for* e *delete,* para que não haja vazamento de memória.

Também, foram implementadas algumas funções acessórias no código-fonte. Visando a prática, foram criadas funções que ordenam as vendas, que buscam por nome e que geram o maior e menor elemento.

e. Erros e dificuldades

Um dos problemas enfrentados foi uma pequena dificuldade de transformar o que foi aprendido em prática, principalmente a parte de *casting* e polimorfismo. Além disso, alguns erros por falta de atenção promoveram algumas horas extras de programação. Também, vale ressaltar a busca por um planejamento ideal para o desenvolvimento do projeto, de um jeito que torne o código mais compreensível e objetivo.

6. Conclusão

O trabalho permitiu o desenvolvimento dos pilares da POO em um projeto maior. Foi possível aprender e praticar muito dos conceitos vistos nas aulas. Foi trabalhoso programar e desenvolver um raciocínio que atendesse a proposta dada, mas ao mesmo tempo também foi muito proveitoso compreender e interligar os conceitos com soluções para os problemas que foram enfrentados. Porém, também em alguns momentos, faltou uma especificação maior

sobre alguns métodos propostos, de modo que a liberdade que foi dada se transformou em uma insegurança de alcançar à expectativa da proposta.