

Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Ciência da Computação

Trabalho Prático de Programação Orientada a Objetos

BCC221 - Programação orientada a objeto

Felipe Braz Marques, Lucas Chagas, Matheus Peixoto, Nicolas Mendes, Pedro
Henrique Rabelo Leão de Oliveira, Pedro Moraes

Professor: Guillermo Camara Chavez

Ouro Preto
30 de junho de 2023

Sumário

1	Introdução	1
1.1	Considerações iniciais	1
1.2	Ferramentas utilizadas	1
1.3	Especificações da máquina	1
1.4	Instruções de compilação e execução	1
2	UML	2
2.1	Cadastro	2
2.2	Hora	3
2.3	Ponto	3
2.4	Venda	3
2.5	Pessoa	4
2.6	Funcionario	4
2.7	Chefe	5
2.8	Supervisor	5
2.9	Vendedor	6
3	Decisões de projeto	7
4	Funcionamento do programa	8
4.1	Login Chefe	8
4.2	Login vendedor	8
4.3	Login Supervisor	8
5	Considerações Finais	9

Lista de Figuras

1	Diagrama UML	2
---	------------------------	---

Lista de Códigos Fonte

1	Classe: Cadastro	2
2	Classe: Hora.	3
3	Classe: Ponto.	3
4	Classe: Venda.	4
5	Classe: Pessoa	4
6	Classe: Funcionario	4
7	Classe Chefe	5
8	Classe: Supervisor.	5
9	Classe: Vendedor.	6

1 Introdução

Neste Trabalho prático será desenvolvido um sistema que permita realizar o cadastro de funcionários de diferentes tipos e realizar o controle de pontos e horas trabalhadas dos funcionários. A implementação foi realizada em C++ utilizando majoritariamente o paradigma de orientação a objeto.

1.1 Considerações iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code.¹
- Linguagem utilizada: C++11.
- Ambiente de desenvolvimento da documentação: Overleaf L^AT_EX.²

1.2 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *Valgrind*: ferramentas de análise dinâmica do código.

1.3 Especificações da máquina

O computador utilizado possui as seguintes especificações:

- Intel Core i5 9^a geração.
- 16GB de RAM.
- Ubuntu 22.1

1.4 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
g++ -Wall Model/*.c Vision/*.cpp -o exe
```

Usou-se para a compilação as seguintes opções:

- *-Wall*: para mostrar todos os possível *warnings* do código.

Para a execução do programa basta digitar:

```
./exe
```

¹Visual Studio Code está disponível em <https://code.visualstudio.com>

²Disponível em <https://www.overleaf.com/>

2 UML

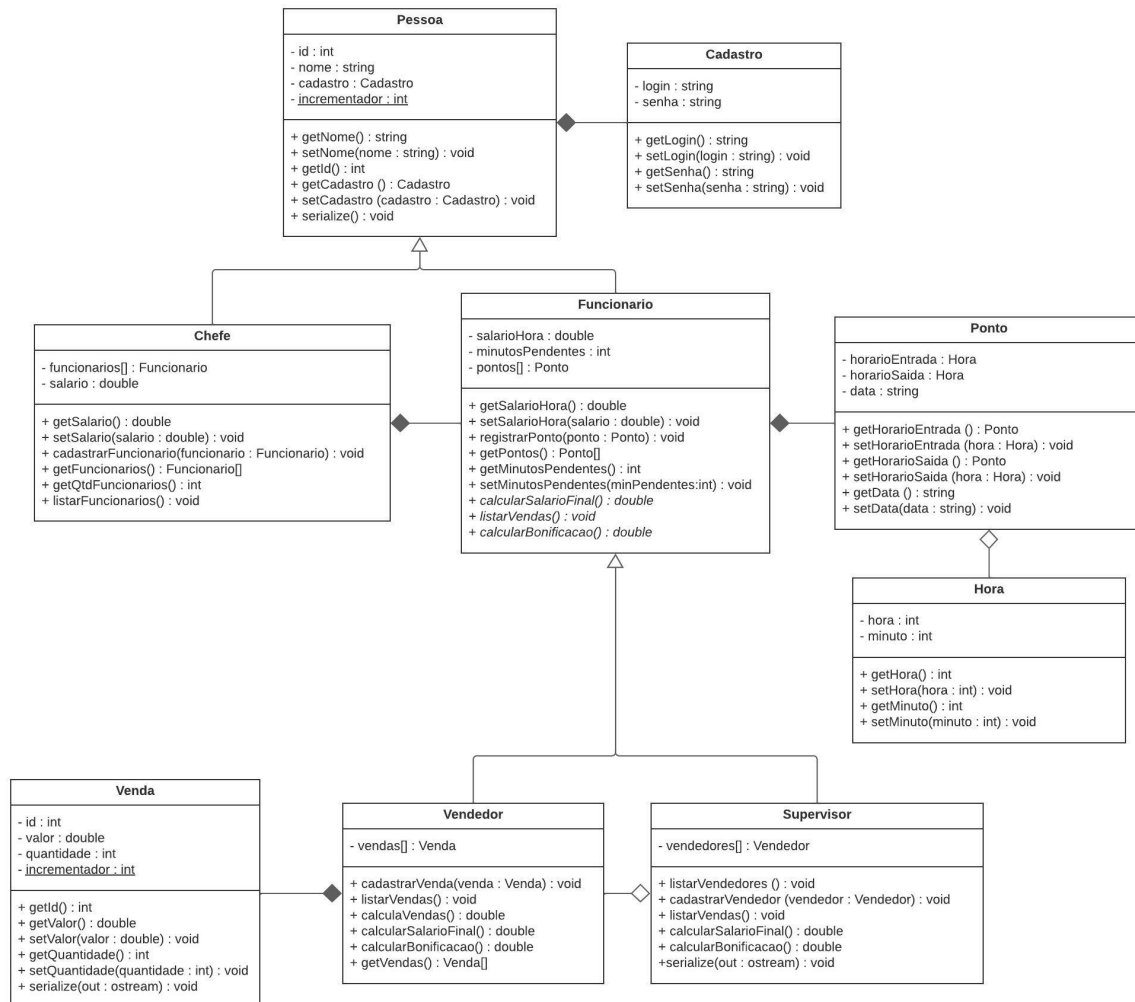


Figura 1: Diagrama UML

A figura representa o diagrama UML que foi utilizado para a realização do trabalho. Assim, os próximos tópicos abordarão as classes, começando por aquelas que funcionam sem necessitar de outras, para as que possuem um pouco mais e terminando com as que de fato necessitam das outras.

2.1 Cadastro

A classe Cadastro é utilizada para facilitar o acesso dos usuários ao sistema a partir do login e senha, uma vez que é possível realizar uma sobrecarga do operador == para comparar o cadastro do usuário e um cadastro temporário que indica os dados que o usuário informou ao tentar acessar o sistema.

```

1
2 class Cadastro{
3     string login;
4     string senha;
5 public:
6     Cadastro(string = "", string="");
  
```

```

7     virtual ~Cadastro();
8     string getLogin() const;
9     void setLogin(string);
10    string getSenha() const;
11    void setSenha(string);
12    friend bool operator==(const Cadastro&, const Cadastro&);
13 };

```

Código 1: Classe: Cadastro

2.2 Hora

A classe Hora possui dois atributos para representar uma hora, sendo um inteiro para hora e outro para minutos. Ademais, possui getters e setters para ambos atributos.

```

1     class Hora{
2         int hora;
3         int minuto;
4     public:
5         Hora(int = 0, int = 0);
6         virtual ~Hora();
7         int getHora() const;
8         void setHora(int) ;
9         int getMinuto() const;
10        void setMinuto(int);
11 };

```

Código 2: Classe: Hora.

2.3 Ponto

A classe Ponto possui uma string para representar a data, que estará no formato DD/MM/YYYY, verificado ao cadastrar um novo ponto, sendo que possui getter e setter.

Também há dois atributos para marcar a entrada e saída, sendo ambos do tipo Hora, que também irá possuir getters e setters.

```

1     class Ponto{
2         Hora * horarioEntrada;
3         Hora * horarioSaida;
4         string data;
5     public:
6         Ponto(Hora * = new Hora(), Hora * = new Hora(), string = "");
7         virtual ~Ponto();
8         Hora * getHorarioEntrada() const;
9         void setHorarioEntrada(Hora *);
10        Hora * getHorarioSaida() const;
11        void setHorarioSaida(Hora *);
12        string getData() const;
13        void setData(string);
14        void serialize(ostream &out) const;
15        friend ostream & operator << (ostream & out, const Ponto &h);
16 };

```

Código 3: Classe: Ponto.

2.4 Venda

A classe Venda possui uma variável estática inteira denominada incrementador, que irá ser responsável por atribuir o valor ao Id da venda e, em sequência, é incrementado em mais um. Ademais, ainda temos os atributos valor e quantidade, que serão definidos pelo usuário ao cadastrar a venda.

```

1 class Venda{
2     int id;
3     static int incrementador; //variavel para setar o id de cada instancia
4     double valor; //valor unitario de cada item
5     int quantidade;
6 public:
7     Venda(double = 0.0, int = 0);
8     virtual ~Venda();
9     int getId() const;
10    double getValor() const;
11    int getQuantidade() const;
12    void setValor(double);
13    void setQuantidade(int);
14    void serialize(ostream&) const;
15    friend ostream& operator<<(ostream&, const Venda&);
16 };

```

Código 4: Classe: Venda.

2.5 Pessoa

A classe Pessoa possui um incrementador estático responsável por definir o Id da pessoa, sendo que o seu valor será definido na instanciação da pessoa e o incrementador terá o seu valor acrescido em um. Ademais, também possui um atributo string nome, além de possuir os getter e setters.

```

1
2 class Pessoa {
3     int id;
4     static int incrementador; //variavel para setar o id de cada instancia
5     string nome;
6     Cadastro * cadastro;
7 public:
8     Pessoa(string="", Cadastro * = new Cadastro());
9     virtual ~Pessoa();
10    string getNome() const;
11    void setNome(string);
12    int getId() const;
13    Cadastro* getCadastro() const;
14    void setCadastro(Cadastro *);
15    void serialize(ostream&) const;
16    friend ostream & operator<< (ostream & out, const Pessoa & obj);
17 };

```

Código 5: Classe: Pessoa

2.6 Funcionario

A classe Funcionário é uma classe abstrata para as próximas classes, que serão Vendedor e Supervisor. Sendo que os atributos exclusivos serão o salarioHora, os minutosPendentes que ainda faltam para usuário para completar as horas mínimas. Também temos um vetor de pontos para a classe Ponto que indicará todos os pontos que o usuário cadastrou.

Temos também os getters e setters para o salarioHora e minutosPendentes, sendo que, para cadastrar os pontos, é feito a partir dos menus do código.

Ademais, vale ressaltar que as funções que tornarão a classe em abstrata são a calcularSalarioFinal, listarVendas e calcularBonificacao, sendo todas virtuais puras.

```

1 class Funcionario : public Pessoa{
2     double salarioHora;
3     int minutosPendentes;
4     vector<Ponto*> pontos;

```

```

5 public:
6     Funcionario(string="", Cadastro* = new Cadastro(), double=0);
7     virtual ~Funcionario();
8     double getSalarioHora() const;
9     void setSalarioHora(double);
10    int getMinutosPendentes() const;
11    void setMinutosPendentes(int);
12    void registrarPonto(Ponto*);
13    vector<Ponto*> getPontos() const;
14    void serialize(ostream&) const;
15    //funcao abstrata, sera utilizada para o polimorfismo
16    virtual double calcularSalarioFinal() = 0;
17    virtual void listarVendas() = 0;
18    virtual double calcularBonificacao() = 0;
19 };

```

Código 6: Classe: Funcionario

2.7 Chefe

A classe Chefe possui como seus atributos um vetor de Funcionario e um salario para o mesmo, tendo getter e setter.

Há também uma função para cadastrarFuncionario, que adiciona um ponteiro de Funcionario ao vetor. A getFuncionarios que retorna o vetor, a getQtdFuncionarios que retorna a quantidade de itens no vetor e a listarFuncionarios que exibe no console todos os funcionarios.

```

1 class Chefe : public Pessoa{
2     vector<Funcionario*> funcionarios;
3     double salario;
4 public:
5     Chefe(string= "", Cadastro* = new Cadastro(), double = 0.0);
6     virtual ~Chefe();
7     double getSalario() const;
8     void setSalario(double);
9     void cadastrarFuncionario(Funcionario*);
10    vector<Funcionario*> getFuncionarios() const;
11    int getQtdFuncionarios();
12    void listarFuncionarios() const;
13 };

```

Código 7: Classe Chefe

2.8 Supervisor

A classe Supervisor herda da classe funcionario e possui um vetor de vendedores como atributo, Esta classe implementa as funções abstratas "calcularSalarioFinal", "calcularBonificacao" além de implementar a função "cadastrarVendedor" que adiciona vendedores no atributo "vendedores" e "listarVendedores" que retorna todos os vendedores adicionados no atributo "vendedores".

```

1 class Supervisor : public Funcionario{
2     vector<Vendedor*> vendedores;
3 public:
4     Supervisor(string="", Cadastro * = new Cadastro(), double=0.0);
5     virtual ~Supervisor();
6     void cadastrarVendedor(Vendedor*);
7     void listarVendedores();
8     void listarVendas();
9     double calcularSalarioFinal();
10    double calcularBonificacao();
11    void serialize(ostream&);
12    friend ostream& operator<<(ostream&, Supervisor&);

```

```
13 };
```

Código 8: Classe: Supervisor.

2.9 Vendedor

A classe Vendedor possui um único atributo exclusivo, um vetor de Venda. Sendo que, para adicionar uma venda é utilizado o método cadastrarVenda, também temos a calculaVendas que retorna o total de dinheiro que possui nas suas vendas. Também temos as funções sobrescritas de Funcionário adequandas para o vendedor, sendo elas: calcularSalarioFinal, calcularBonificacao e listarVendas.

```
1 class Vendedor : public Funcionario{
2     vector<Venda*> vendas;
3 public:
4     Vendedor(string="", Cadastro* = new Cadastro(), double=0);
5     virtual ~Vendedor();
6     void cadastrarVenda(Venda*);
7     void listarVendas();
8     double calculaVendas();
9     double calcularBonificacao();
10    double calcularSalarioFinal();
11    vector<Venda*> getVendas();
12    void serialize (ostream&);
13    friend ostream& operator<<(ostream&, Vendedor&);
14 };
```

Código 9: Classe: Vendedor.

3 Decisões de projeto

Após a construção da UML, com a criação das classes e estabelecendo o tipo de associação entre elas, definiu-se algumas resoluções do projeto, visando uma implementação harmônica com os requisitos do TP.

Primeiramente, o sistema foi desenvolvido para o uso de um único chefe, que já é setado na main com `usuario = admin` e `senha = admin`.

Ademais, todas as informações referentes à jornada de trabalho (horas trabalhadas, pendentes e extras) e os cálculos envolvendo salário foram feitos com minutos, tendo em vista que todos os inputs, feitos pelo user, que envolvem tempo, contém o horário com horas e minutos.

Além disso, o sistema sempre trabalha com uma jornada de trabalho mensal para os funcionários, estipulada em 160 horas (8 horas diárias por 5 dias na semana [4 semanas por mês]), a qual foi definida na constante `"CARGA_HORARIA_MENSAL"` no `"Funcionario.h"`, e utiliza o atributo `minutosPendentes` para realizar cálculos relacionados a jornada de trabalho e salário final, como horas pendentes, horas extras e horas trabalhadas. Ao instanciar qualquer objeto da classe `Funcionario`, o atributo `minutosPendentes` recebe o valor 9600 minutos (correspondentes a jornada de trabalho mensal, 160 horas). No momento em que o funcionário registra um ponto, o tempo de trabalho é calculado e subtraído de `minutosPendentes`.

Portanto, a lógica é a seguinte: Se o valor de `minutosPendentes` é positivo, significa que o valor de `minutosPendentes` representa o tempo que falta para atingir a carga horária mensal estipulada, $(9600 [\text{carga horária mensal estipulada}] - \text{minutosPendentes})$ representa o tempo trabalhado e que não há nenhuma hora extra.

Se o valor de `minutosPendentes` é nulo, significa que o tempo trabalhado é igual a 9600 minutos (carga horária mensal estipulada), ou seja, o tempo de trabalho pendente é nulo e não há nenhuma hora extra.

Se o valor de `minutosPendentes` é negativo, significa que o funcionário trabalhou um tempo superior à 9600 minutos (carga horária mensal estipulada), e, portanto, para o seu salário é considerado os 9600 minutos de trabalho mensal e as horas extras, que tem seu valor como o módulo de `minutosPendentes`.

Também, no que se refere ao cálculo do salário envolvendo o valor por hora, esse mesmo valor é dividido por 60, para representar o valor por minuto no cálculo que trabalha com minutos. Todos os cálculos referentes ao salário final dos funcionários são feitos na função `calcularSalarioFinal()` presente na classe dos funcionários como um método abstrato, e implementado na classe dos vendedores e dos supervisores.

Ainda, nas classes em que possuem `Vectors` em seus atributos, acordou-se que os mesmos seriam de ponteiros para uma classe, pois, desse modo, pode-se trabalhar com a alocação dinâmica, uma estratégia adotada para evitar um uso exarcebado de memória (caso fosse instanciado objetos estaticamente e feito cópias de seus atributos para outros objetos da mesma classe). Inicialmente, optamos por utilizar a passagem por referência dos objetos das classes nas funções (realizando alocação estática dos objetos dentro das funções), entretanto, ao finalizar a execução das funções, ao sair do seu escopo esses objetos eram destruídos e as referências passadas eram perdidas. Portanto, em função desse empecilho, optamos pela alocação dinâmica e a passagem de referência via ponteiros.

4 Funcionamento do programa

Ao iniciar o programa, podemos escolher entre duas opções, realizar um Login (1), ou sair do sistema (2).

Realizando o login, podemos escolher entre fazer um login como chefe (1) ou funcionário (2). Sendo que, o código já possui três usuários cadastrados, um chefe com login "admin" e senha "admin"; Um supervisor com login "marcelo" e senha "123" & Um vendedor com login "joao" e senha "123".

4.1 Login Chefe

Realizando o login com o chefe, é possível Cadastrar Funcionário (1); Listar Funcionários (2); Checar Ponto dos Funcionários (3); Cálculo de salários e bonificações (4); Retornar a tela inicial (5).

Cadastrando um funcionário, podemos cadastrar um supervisor ou um vendedor. Cadastrando um supervisor, é solicitado o nome, login, senha e salário hora. Já para cadastrar um vendedor, também são solicitados os mesmos itens, mas também é solicitado o Id de um supervisor, pois o vendedor será inserido no vetor desse supervisor.

Listando os funcionários, teremos um resumo sobre todos, obtendo a sua função, nome, id, login, salário por hora e bonificações.

Checar o ponto dos funcionários irá exibir, de forma ordenada, dos funcionários que faltam mais tempo para obter o tempo mínimo para os que menos faltam. Além de mostrar o tempo trabalhado e o pendente.

Calcular o salário e bonificações mostra o nome dos funcionários, seus salários finais e bonificações, a partir da ordem do mais antigo a ser cadastrado até o mais recente.

Por fim, retornar à tela inicial abre novamente a escolha de login e sair do sistema.

4.2 Login vendedor

Realizando o login como vendedor, é possível cadastrar ponto (1), exibir salário (2), cadastrar venda (3), listar vendas (4) e retornar à tela inicial (5).

Cadastrar ponto solicita ao usuário uma data no formato DD/MM/YYYY, por meio de uma string. Depois é solicitado a hora e minuto da entrada e, depois, a hora e minuto da saída.

Exibir salário irá exibir o salário final e a bonificação do vendedor.

Cadastrar venda, solicita a quantidade de itens vendidos e o preço unitário de cada.

Listar Vendas exibe o Id da venda, a quantidade de itens da venda e o valor unitário de cada item.

Por fim, retornar à tela inicial abre novamente a escolha de login e sair do sistema.

4.3 Login Supervisor

Realizando o login como supervisor, é possível cadastrar ponto (1), exibir salário (2), listar vendas (3) e retornar à tela inicial (4).

Cadastrar ponto e exibir salário terá o mesmo comportamento das funções para o vendedor.

A função de listar vendas irá listar todas as vendas dos vendedores que estão no vetor de vendedores do funcionário.

Por fim, retornar à tela inicial abre novamente a escolha de login e sair do sistema.

5 Considerações Finais

Com o fim do trabalho, foi possível compreender mais sobre o desenvolvimento de programas utilizando o paradigma da programação orientada a objetos, utilizando desde propriedades básicas como o relacionamento entre classes até polimorfismo, downcasting e exceções.

Em geral, um dos maiores problemas que o grupo teve durante a execução do trabalho foi em relação à utilização de ponteiros para instanciação de objetos. Um exemplo foi na main, onde foi necessário iniciar um objeto chamado `funcionarioLogado` com ponteiro nulo, onde, somente na função de verificar login, um objeto é atribuído a ele. Dessa forma foi necessário passar os dados por "ponteiro de ponteiro".

Por fim, pudemos aplicar, sem grande dificuldade, os conceitos apresentados em sala de aula, como o polimorfismo de funcionários e o fato dela ser uma classe abstrata. A utilização de downcasting para acessar os métodos exclusivos de vendedor e supervisor. A utilização de composição para a classe Pessoa e Cadastro. Também temos a herança vinda da superclasse Pessoa para Chefe e Funcionário, onde Vendedor e Supervisor estendem da mesma.