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

Sistema de Gerenciamento de Vendas

BCC221 - Programação Orientada a Objetos

Bruno Alves Braga
Jéssica Machado
Kézia Alves Brito
Vitor Oliveira Diniz
Professor: Guillermo Cámara-Chávez

Ouro Preto
29 de junho de 2023

Sumário

Lista de Figuras

1 Introdução

Neste trabalho, foi necessário entregar o código em C++ e um relatório referente ao que foi desenvolvido. O objetivo deste é implementar um sistema que permita realizar o cadastro de funcionários e o controle de pontos dos mesmos, seguindo algumas regras especificadas no documento do Trabalho Prático.

1.1 Especificações do problema

O gerenciamento dos funcionários espera uma alta coesão entre as classes para um bom aproveitamento do código e de suas funções. Sendo assim, por ser um sistema conciso, devemos documentar bem todas essas relações entre as diferentes classes e fazer uma implementação de modo que uma classe não assuma responsabilidades que não são suas.

Assim, devemos desenvolver um programa, em linguagem C++, que nos permita cadastrar, logar, listar, entre outras operações envolvendo dados de funcionários fornecidos pelo usuário, respeitando as restrições impostas pelo sistema, como: horas máximas diárias e semanais e bonificações.

1.2 Considerações iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code + *LiveShare*.
- Linguagem utilizada: C++.
- Compilador utilizado: g++ (GCC) *version* 13.1.1 20230429.
- Ambiente de desenvolvimento da documentação: Overleaf LaTeX.

1.3 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

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

1.4 Especificações da máquina

As máquinas onde o desenvolvimento e os testes foram realizados possuem as seguintes configurações:

- **Computador:** Vitor.
- Processador: Ryzen 7-5800H.
- Memória RAM: 16 GB.
- Sistema Operacional: Arch Linux x86_64.
- **Computador:** Kézia.
- Processador: Intel Core i5.
- Memória RAM: 8 GB.
- Sistema Operacional: Manjaro Linux.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
make
```

Usou-se para a compilação as seguintes opções:

- *-c*: para compilar o arquivo sem vincular os arquivos do tipo objeto.
- *-o*: para vincular os arquivos do tipo objeto.
- *-Wall*: para mostrar todos os possíveis *warnings* do código.
- *-g*: para compilar com informação de depuração e ser usada pelo *Valgrind*.

Para a execução do projeto, basta digitar:

Executando o projeto

```
./sistema
```

2 Implementação

Inicialmente, foi feito um esboço preliminar de um diagrama UML para que pudéssemos ter um molde das relações interclasse, a fim de realizar uma implementação sucinta. Como sabíamos que, com o tempo, a necessidade de novas classes iria aparecer, atualizamos o esboço conforme essa necessidade, para depois criar o diagrama final. A partir disso, começamos desenvolvendo as classes mais baixas hierárquicamente, para que, a cada etapa, pudéssemos testar suas funcionalidades e garantir a coesão do programa.

2.1 Recursos Utilizados

2.1.1 Herança

A herança, princípio próprio da Programação Orientada a Objetos, permite criar uma nova classe a partir de outra já existente, herdando atributos e métodos desta. Assim, podemos reaproveitar blocos de código, que seriam inteiramente copiados por classes semelhantes, através de classes bases e aumentar a especialização a cada herança, obtendo, assim, uma ótima representação de problemas do mundo real em código.

2.1.2 Sobrecarga

A sobrecarga é um mecanismo utilizado para descrever como classes se comportam mediante o uso de determinados operadores sobre as mesmas. O programa é incapaz, por exemplo, de realizar operações como adição (+) sobre classes, a não ser que seu funcionamento seja estritamente definido pelo programador previamente.

2.1.3 Exceções

Como o programa depende de algumas regras para seu funcionamento, uma das primeiras funcionalidades pensadas e implementadas foram as exceções, em que, se alguma coisa fere uma regra básica do funcionamento do sistema, iremos lançar uma exceção para contar ao usuário o que aconteceu, sempre lembrando de tratar a exceção de maneira adequada para que o programa não interrompa seu fluxo e deixe de funcionar em um momento crítico, melhorando, assim, o fluxo de execução de nosso programa. Além de utilizar exceções já implementadas nas bibliotecas padrões do C++, como de *input* inválido, criamos algumas exceções específicas, por exemplo, para o caso de o funcionário ultrapassar as horas permitidas de trabalho.

2.1.4 *Vector*

Vectors são estruturas de dados homogêneas capazes de armazenar valores, indexando-os em ordem crescente na memória. Diferentemente dos vetores normais alocados dinamicamente, eles realizam a realocação de espaço na memória automaticamente, facilitando o uso e possibilitando o armazenamento de uma quantidade "indeterminada"de dados (memória RAM do usuário). Já que nesse sistema temos variantes sem restrição pré-definida de quantidade, como as vendas, é fundamental utilizarmos estruturas desse tipo.

3 Arquitetura do Programa

3.1 Classes Utilizadas

3.1.1 Pessoa

Como descrito pelo arquivo do Trabalho Prático, decidimos começar a implementação pela classe *Pessoa*, que seria a unidade básica de *Chefe* e de *Funcionario*. Assim, como uma pessoa é pouco especializada e reflete pouco do sistema, sua inicialização se deu apenas pelo nome.

3.1.2 Chefe

Para o chefe, como é uma especialização bem específica, precisamos definir todos os seus atributos que serão utilizados posteriormente, como um *vector* de funcionários, usuário, senha e salário, afinal, o chefe controla todos os funcionários através de suas diferentes funções. Para isso, criamos métodos que permitem essa manipulação e controle de dados mais específicos, como codastrar os funcionários.

3.1.3 Funcionário

Já *Funcionario*, como é uma classe abstrata, definimos atributos que poderiam ser utilizados pelas duas classes que os herdariam, *Supervisor* e *Vendedor*. Seus métodos, além dos *setters* e *getters*, também incluem cadastrar ponto, calcular horas semanais e calcular horas mensais.

3.1.4 Supervisor

Aqui temos uma especialização de *Funcionario*, em que, além de suas funções básicas, que foram adquiridas através de uma herança, tivemos que redefinir alguns métodos que possuem um funcionamento mais específico, como calcular salário, e temos um *vector* de vendedores, afinal, o supervisor precisa dessa interação em alguns de seus métodos.

3.1.5 Vendedor

Assim como *Supervisor*, temos uma outra especialização de *Funcionario*, onde realizamos os mesmos procedimentos, redefinindo alguns métodos e adicionando um *vector* de vendas, pois o vendedor precisa das mesmas para calcular algumas bonificações de seu salário.

3.1.6 Venda

A venda, como citado anteriormente, possui um valor e uma data, pois, assim, podemos calcular o salário mensal do funcionário de forma consistente e cadastrar as vendas em seu *vector* para auxiliar o funcionamento interclasse do sistema.

3.1.7 Data

A classe *Data*, como o próprio nome indica, representa uma data do ano, contendo valores inteiros que indicam o dia, mês e ano. Ela é utilizada como referência para realizar a descrição do ponto.

3.1.8 Horário

O horário irá se integrar diretamente com o ponto, pois os funcionários dependem do horário de entrada e saída para podermos calcular seu ponto.

3.1.9 Ponto

O ponto está presente na classe dos funcionários e tem como finalidade controlar o período de trabalho do funcionário, para que, assim, possamos imprimir seu salário corretamente e verificar se há alguma pendência.

3.1.10 Menu

O menu é onde toda a troca de informações das classes vão ocorrer. Nele há o primeiro *login* de um chefe, e assim ele pode cadastrar vários usuários que irão usufruir do sistema. O menu é responsável por todo o fluxo de execução do nosso programa, mostrando as opções e recebendo todos os *inputs* necessários em cada sub-menu, como cadastro de usuário, cadastro de ponto, entre outros. Aqui também ocorre o tratamento de todas as exceções sempre que há possibilidade de falha no código por não obediência a uma regra definida pelo sistema. Temos como atributos um chefe, que foi previamente cadastrado no sistema, o funcionário *logado* atualmente, caso seja um, e uma variável de controle para que possamos mostrar o menu adequado ao cargo.

3.2 Diagrama UML

Após uma breve explicação de cada classe:

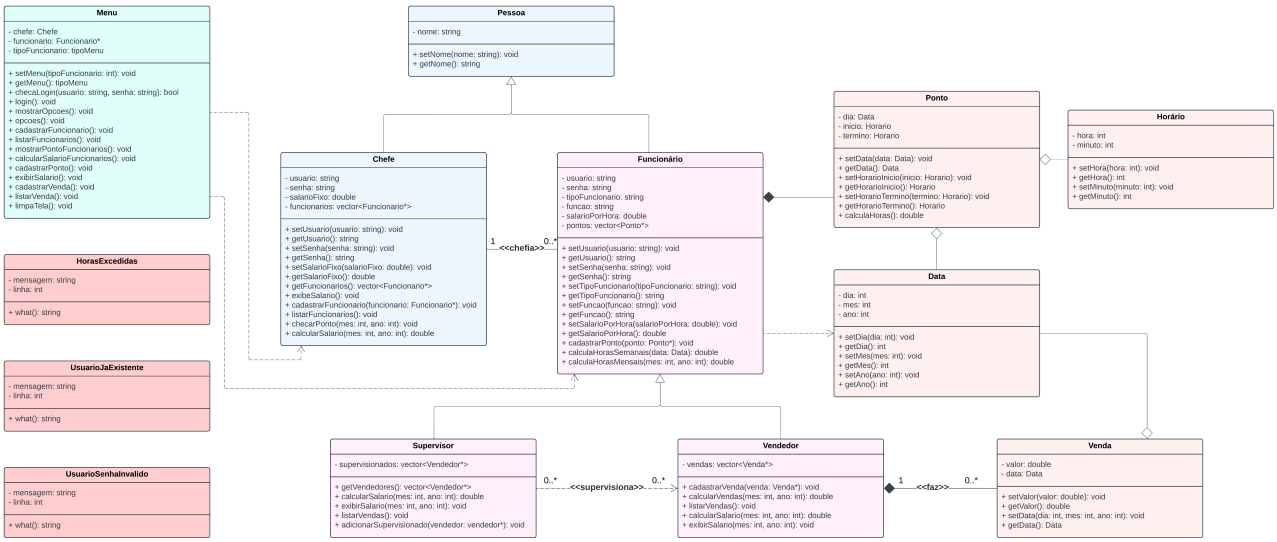

Figura 1: Diagrama UML representando toda a relação interclasse do programa.

Link para melhor visualização do Diagrama UML

3.3 Decisões

Como o comportamento de alguns métodos não estavam descritos especificamente, algumas decisões arbitrárias foram tomadas pela equipe. A maior delas foi que decidimos separar as vendas e pontos através de datas, pois, assim, poderíamos ter um controle melhor do tempo de cada ponto, obedecendo às restrições. Através desse controle de tempo, foi decidido que, assim como acontece na vida real, o salário dos funcionários seria dado mensalmente, dessa forma, devemos sempre inserir mês e ano para procurar o salário e/ou exibi-lo na tela. Com isso, podemos representar melhor o problema nos moldes reais e ter uma noção do quanto de caixa será necessário para pagar os funcionários.

Como também não havia explicitada a dinâmica de um supervisor, decidimos que todo vendedor cadastrado será supervisionado por todos os supervisores já existentes, assim temos uma dinâmica de fácil implementação, mas dispomos de todos os meios para alterar facilmente o código devido à sua alta coesão. Outra decisão foi que, como o chefe não tinha a opção de cadastrar ponto em seu menu, decidimos por adicionar um salário fixo para ele, assim não teríamos dificuldades em calcular seu salário.

Para facilitar o teste do sistema, criamos alguns usuários pré-cadastrados, como um chefe que tem o usuário admin e a senha admin. Esses usuários podem ser consultados na implementação do menu.

4 Análise

Após a finalização e o período de testes do projeto, percebemos algumas coisas que poderiam ter sido melhor implementadas, de forma a facilitar o processo de codificação e suavizar o fluxo de execução do programa.

Poderíamos, por exemplo, ter usado *set*, uma estrutura da *STL* que tem uma inserção ordenada. Esse container pode ser útil em futuras implementações, ajudando a controlar as inserções na medida em que impede duplicatas.

Além disso, teria sido interessante aderir ao uso do container *multiset*, também da *STL*, caso o número de funcionários cadastrados fosse extenso. Sua estruturação em árvore levaria o código a uma complexidade melhor.

5 Conclusão

A partir do desenvolvimento do Sistema de Gerenciamento de Vendas proposto, fomos capazes de entender melhor e fixar os conhecimentos sobre a POO, tais como o uso de classes para representar situações e problemas encontrados na vida real.

Durante a criação do código e a implementação das classes, encontramos certas dificuldades relacionadas à alocação de memória e utilização da estrutura *vector*, que resultou em falhas de segmentação e saídas inesperadas. Com ajuda do professor, reformulando o código e utilizando outros comandos como o *new*, fomos capazes de superar essa barreira e produzir o resultado que queríamos.

Além das complexidades envolvendo manipulação de memória, tivemos empecilhos ao tentar desenvolver classes e funções polimórficas na intenção de evitar repetição de código. Ao decorrer do trabalho, no entanto, conseguimos entender melhor o conceito e resolver o problema.

Dessa forma, foi possível implementar o Sistema de Gerenciamento de Vendas eficientemente, de maneira relativamente simples contando com nosso conhecimento prévio da linguagem de programação C, que muito se assemelha à linguagem utilizada neste trabalho, C++.