

UNIVERSIDADE DE SÃO PAULO

Instituto De Ciências Matemáticas e de Computação

Departamento de Ciências de Computação

Trabalho Final
Sistema de Restaurante

Carlos Henrique Hannas de Carvalho n^o USP: 11965988

Henrique Carobolante Parro n^o USP: 11917987

Lucca Tommaso Monzani n^o USP: 5342324

Prof. Andre de Freitas Smaira

SCC0604 - Programação Orientada a Objetos

16 de dezembro de 2024

Sumário

1	Descrição do Projeto	1
2	Diagramas de Casos de Uso e Classes	1
2.1	Casos de Uso	1
2.1.1	Especificações Textuais dos Casos de Uso	2
2.2	Diagrama de Classes	3
2.2.1	Especificações Textuais do Diagrama de Classes	3
3	Planejamento e Estrutura	7
4	Implementação e Funcionalidades	8
4.1	<i>restaurant.cpp</i>	9
4.2	Pasta <i>header</i>	9
4.2.1	<i>cardapio.hpp</i>	9
4.2.2	<i>cliente.hpp</i>	9
4.2.3	<i>funcionario.hpp</i>	9
4.2.4	<i>mesa.hpp</i>	9
4.2.5	<i>pagamento.hpp</i>	9
4.2.6	<i>pedido.hpp</i>	9
4.2.7	<i>reserva.hpp</i>	9
4.3	Pasta <i>source</i>	9
4.3.1	<i>cardapio.cpp</i>	9
4.3.2	<i>cliente.cpp</i>	10
4.3.3	<i>funcionario.cpp</i>	10
4.3.4	<i>mesa.cpp</i>	10
4.3.5	<i>pagamento.cpp</i>	10
4.3.6	<i>pedido.cpp</i>	10
4.3.7	<i>reserva.cpp</i>	10
5	Guia de Uso e Exemplos	10
5.1	Guia de Uso	10
5.2	Exemplos	11
6	Desafios e Soluções	13
6.1	Sugestões de Projeto	14
A	Implementação	15

Lista de Figuras

1	Diagrama de Casos de Uso.	1
2	Diagrama de Classes.	3
3	Organização do diretório.	8
4	Comando <i>make</i> para compilar a simulação.	10
5	Comando <i>make run</i> para executar a simulação.	11
6	Exemplo da instanciação de dois objetos da classe Cliente.	11
7	Exemplo da instanciação de dois objetos da classe Mesa.	12
8	Exemplo da instanciação de um objeto da classe Reserva.	12
9	Exemplo da instanciação de um objeto da classe Cardápio.	12
10	Exemplo da instanciação de objetos da classe Funcionário.	13
11	Exemplo da instanciação de um objeto da classe Pedido.	13

Listings

1	Arquivo <i>restaurant.cpp</i>	15
2	Arquivo <i>cardapio.hpp</i>	19
3	Arquivo <i>cardapio.cpp</i>	19
4	Arquivo <i>cliente.hpp</i>	20
5	Arquivo <i>cliente.cpp</i>	21
6	Arquivo <i>funcionario.hpp</i>	21
7	Arquivo <i>funcionario.cpp</i>	23
8	Arquivo <i>mesa.hpp</i>	24
9	Arquivo <i>mesa.cpp</i>	24
10	Arquivo <i>pagamento.hpp</i>	25
11	Arquivo <i>pagamento.cpp</i>	26
12	Arquivo <i>pedido.hpp</i>	33
13	Arquivo <i>pedido.cpp</i>	34
14	Arquivo <i>reserva.hpp</i>	35
15	Arquivo <i>reserva.cpp</i>	36

1 Descrição do Projeto

O objetivo da prática é desenvolver um sistema de gerenciamento de restaurantes, voltado para atender uma demanda da sociedade: facilitar diversas operações em estabelecimentos gastronômicos, como já acontece em alguns lugares atualmente. A gerência de um restaurante envolve diversos componentes, como a cozinha, o salão principal de atendimento, o setor financeiro e entre outros - a automatização, através de um sistema, otimiza e facilita o processo de gestão de todos esses fatores mencionados. Portanto, procura-se modelar e implementar um sistema, que seja capaz de envolver os ambientes e pessoas que frequentam o restaurante, a fim de contribuir para a organização e eficiência na coordenação e gestão que o ambiente exige.

Através de estudos e análises em alguns ambientes voltados para gastronomia, identificaram-se alguns atores principais do sistema. São exemplos deles: clientes que frequentam o restaurante, funcionários do restaurante, como chefes de cozinha, caixas, recepcionistas e garçons. Os atores listados colaboram e interagem para o funcionamento do restaurante, com as respectivas funcionalidades.

Associa-se, ao menos uma, funcionalidade para cada perfil de usuário, que interage com o sistema. Alguns exemplos das principais funcionalidades são listadas a seguir: usuários como clientes e caixas relacionam-se em componentes de pagamentos, tendo que realizar um determinado pagamento ou devolução de troco; usuários como clientes e recepcionistas comunicam entre si para realizar, cancelar e verificar o *status* de uma reserva; clientes e garçons estão atrelados aos pedidos: anotar um pedido e entregá-lo ao cliente; funcionários de cozinha, como *chef*, devem preparar o prato.

Na atual entrega do projeto pode-se identificar e modelar os principais componentes que envolvem um sistema de restaurantes, como o gerenciamento na cozinha e atendimentos ao cliente. Em resumo, as modelagens de atores e funcionalidades no diagrama de casos de uso e a modelagem de classes que compõem os restaurantes se mostraram concisas em relação aos requisitos iniciais do sistema - isso possibilita uma base sólida para prosseguir o desenvolvimento e implementação de um sistema conciso de gestão de restaurantes.

2 Diagramas de Casos de Uso e Classes

2.1 Casos de Uso

A figura 1 apresenta o diagrama de casos de uso do sistema de restaurantes:

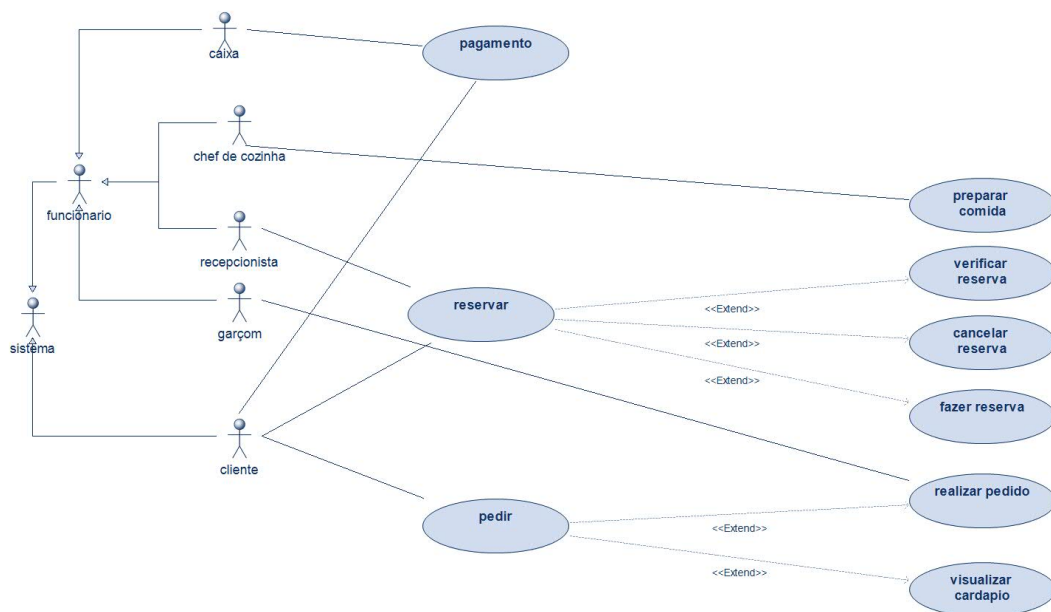


Figura 1: Diagrama de Casos de Uso.

A descrição textual do diagrama de casos de uso da imagem 1 é apresentada na seção 2.1.1.

2.1.1 Especificações Textuais dos Casos de Uso

Caso de Uso: Pedir

Ator Primário: Cliente;

Descrição: O cliente solicita, ao garçom, para visualizar o cardápio ou realizar o pedido final.

Caso de Uso: Pedir [Realizar Pedido]

Ator Primário: Cliente;

Descrição: O cliente faz o pedido do prato. O sistema registra e informa à cozinha para que a comida seja preparada;

Ator Secundário: Garçom;

Descrição: O garçom deve ser capaz de anotar o pedido realizado pelo cliente.

Caso de Uso: Pedir [Visualizar Cardápio]

Ator Primário: Cliente;

Descrição: O cliente visualiza o cardápio disponível, para verificar as opções dos itens e seus respectivos preços.

Caso de Uso: Reservar

Ator Primário: Cliente;

Descrição: O cliente faz uma solicitação de reserva: realizar, verificar o *status* ou cancelar uma reserva;

Ator Secundário: Recepcionista;

Descrição: O recepcionista deve ser capaz de auxiliar o cliente em relação às funcionalidades de reserva.

Caso de Uso: Reservar [Verificar Reserva]

Ator Primário: Cliente;

Descrição: O cliente consulta algumas características da reserva, como *status*, horário, quantidade de pessoas, nome e número da reserva.

Caso de Uso: Reservar [Cancelar Reserva]

Ator Primário: Cliente;

Descrição: O cliente cancela uma reserva previamente feita. O sistema altera o *status* da reserva para *false*.

Caso de Uso: Reservar [Fazer Reserva]

Ator Primário: Cliente;

Descrição: O cliente pode realizar uma reserva no restaurante, especificando alguns dados como data/horário, nome do cliente e quantidade de pessoas.

Caso de Uso: Preparar Comida

Ator Primário: *Chef* de cozinha;

Descrição: O *chef* de cozinha deve preparar a comida do pedido.

Caso de Uso: Pagamento

Ator Primário: Cliente;

Descrição: O cliente deve realizar o pagamento do que foi consumido do restaurante;

Ator Secundário: Caixa;

Descrição: O caixa deve ser responsável por gerir o pagamento feito pelo cliente.

2.2 Diagrama de Classes

A figura 2 apresenta o diagrama de classes do sistema de restaurantes:

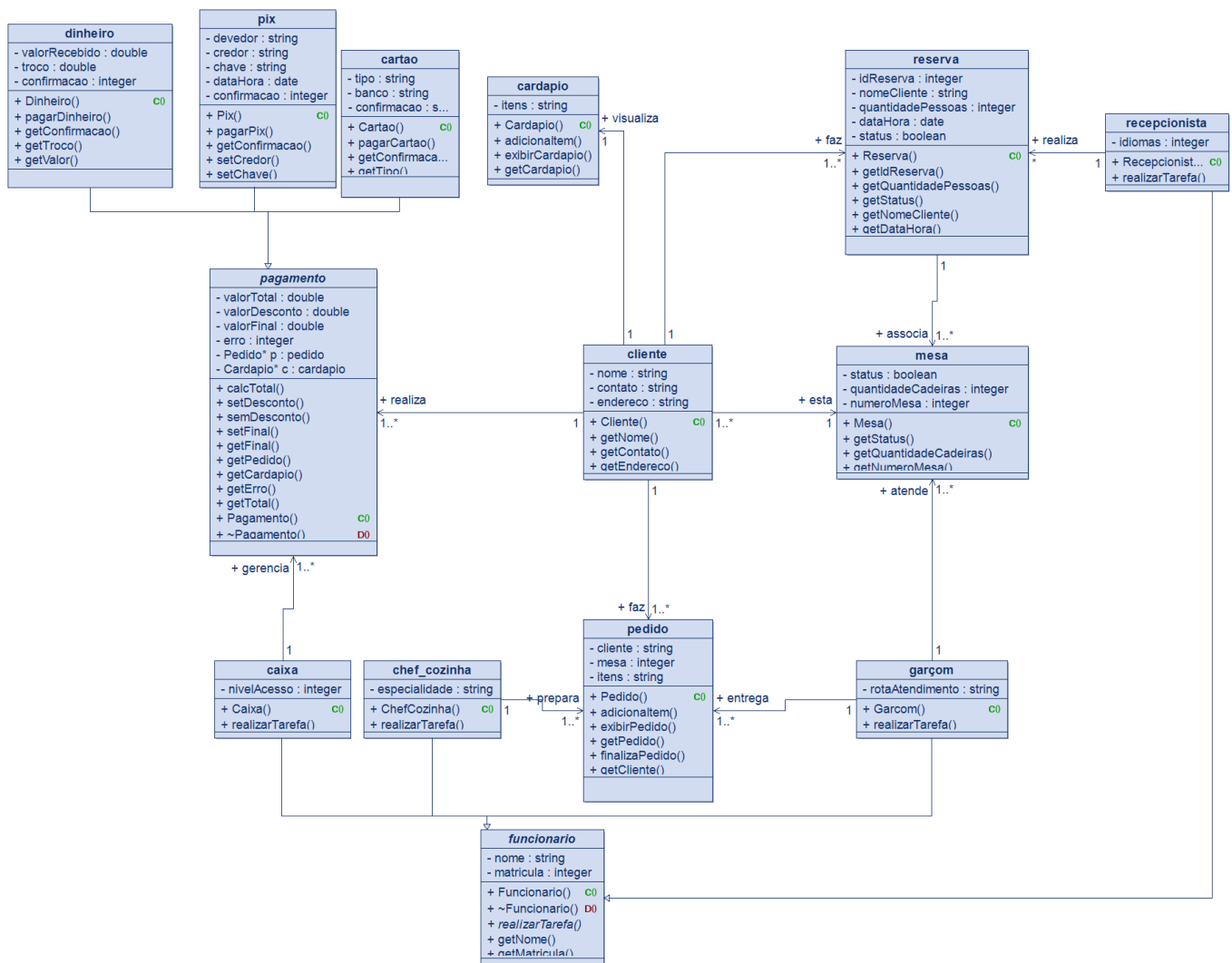


Figura 2: Diagrama de Classes.

A descrição textual do diagrama de classes, incluindo os respectivos atributos e métodos, da imagem 2 é apresentada na seção 2.2.1.

2.2.1 Especificações Textuais do Diagrama de Classes

Abaixo há a descrição das classes do sistema, incluindo os respectivos atributos e métodos. Conforme a figura 2, os atributos descritos são privados.

Classe Funcionário (classe abstrata)

Atributos: nome e matricula;

- nome: *string* com o nome do funcionário;
- matricula: número inteiro, que representa o número de matrícula (registro) do funcionário.

Métodos: Funcionario, ~Funcionario, realizarTarefa, getNome e getMatricula;

- Funcionario: construtor da classe Funcionário;
- ~Funcionario: destrutor da classe Funcionário;
- realizarTarefa: método polimorfo da classe Funcionário;
- getNome: retorna o nome do funcionário;

- getMatricula: retorna a matrícula do funcionário.

Classe Recepcionista

Atributos: idiomas;

- idiomas: número inteiro, que representa a quantidade de idiomas que recepcionista fala.

Métodos: Recepcionista e realizarTarefa

- Recepcionista: construtor da classe Recepcionista;
- realizarTarefa: retorna a atividade de recepcionista.

Classe Garçom

Atributos: rotaAtendimento;

- rotaAtendimento: *string* que representa a rota de atendimento do garçom dentro do restaurante.

Métodos: Garcom e realizarTarefa;

- Garcom: construtor da classe Garçom;
- realizarTarefa: retorna a atividade de garçom.

Classe Chef de Cozinha

Atributos: especialidade;

- especialidade: *string* que representa a especialidade de prato do *chef* de cozinha.

Métodos: ChefCozinha e realizarTarefa;

- ChefCozinha: construtor da classe ChefCozinha;
- realizarTarefa: retorna a atividade de chef de cozinha.

Classe Caixa

Atributos: nivelAcesso;

- nivelAcesso: número inteiro que representa o nível de acesso ao caixa (máquina registradora) do funcionário.

Métodos: Caixa e realizarTarefa;

- Caixa: construtor da classe Caixa;
- realizarTarefa: retorna a atividade de caixa.

Classe Pagamento (classe abstrata)

Atributos: valorTotal, valorDesconto, valorFinal, erro, pedido e cardapio;

- valorTotal: número *double* com o valor total da conta;
- valorDesconto: número *double* com o valor de desconto;
- valorFinal: número *double* com o valor final da conta ($\text{valorFinal} = \text{valorTotal} - \text{valorDesconto}$);
- erro: variável auxiliar;
- pedido: ponteiro para o pedido;
- cardapio: variável auxiliar para verificar o preço do cardápio para realizar o pagamento.

Métodos: calcTotal, setDesconto, semDesconto, setFinal, getFinal, getPedido, getCardapio, getErro, getTotal, Pagamento e ~Pagamento;

- Pagamento: construtor da classe Pagamento;
- ~ Pagamento: destrutor da classe Pagamento;
- calcTotal: calcula o valor total da conta;
- setDesconto: Adiciona desconto ao preço;
- semDesconto: pagamento final sem desconto;
- setFinal: pagamento final com desconto;
- getFinal: retorna o valor final da conta;
- getPedido: retorna um ponteiro para o pedido;
- getCardapio: retorna a classe do cardapio;
- getErro: retorna caso der erro;
- getTotal: calcula o valor total.

Classe Dinheiro

Atributos: valorRecebido, troco e confirmacao;

- valorRecebido: número *double* que representa o valor pago em dinheiro;
- confirmacao: *status* da confirmação;
- troco: número *double* que representa o troco, caso houver, a ser recebido pelo cliente.

Métodos: Dinheiro, pagarDinheiro, getConfirmacao, getTroco e getValor;

- Dinheiro: construtor da classe Dinheiro;
- pagarDinheiro: método interativo para realizar pagamento em dinheiro;
- getConfirmacao: retorna *status* da confirmação;
- getTroco: retorna o valor do troco;
- getValor: retorna o valor recebido pelo cliente.

Classe Pix

Atributos: devedor, credor, chave e confirmacao;

- devedor: *string* com o nome do cliente que realizará o pix;
- credor: *string* com o nome do restaurante (receberá o pix);
- chave: *string* com a chave pix do restaurante;
- confirmacao: *status* da confirmação.

Métodos: Pix, pagarPix, getConfirmacao, setCredor e setChave;

- Pix: construtor da classe Pix;
- pagarPix: método interativo para realizar pagamento via Pix;
- getConfirmacao: retorna *status* da confirmação;
- setCredor: retorna o credor do Pix;
- setChave: retorna a chave Pix do credor.

Classe Cartão

Atributos: confirmacao, tipo e banco;

- confirmacao: *status* da confirmação;
- tipoPagamento: representa um cartão de débito ou crédito;

- banco: *string* com o nome do banco em que o cartão está associado.

Métodos: Cartao, pagarCartao, getConfirmacao e getTipo;

- Cartao: construtor da classe Cartão;
- pagarCartao: método interativo para realizar pagamento com cartão;
- getConfirmacao: retorna *status* da confirmação;
- getTipo: retorna a opção do tipo de pagamento.

Classe Pedido

Atributos: cliente, mesa e itens;

- cliente: *string* com o nome do cliente que fez o pedido;
- mesa: número da mesa em que o cliente está alocado;
- itens: vetor de *string*, com os pares de item e quantidade do item.

Métodos: Pedido, adicionarItem, exibirPedido, getPedido, finalizarPedido e getCliente;

- Pedido: construtor da classe Pedido;
- adicionarItem: adiciona o item e a quantidade do item ao pedido;
- exibirPedido: exibe o pedido do cliente;
- getPedido: retorna itens do pedido;
- finalizarPedido: finaliza o pedido;
- getCliente: retorna o nome do cliente que fez o pedido.

Classe Cliente

Atributos: nome, contato e endereco;

- nome: *string* com o nome do cliente;
- contato: *string* com o contato do cliente;
- endereco: *string* com o endereço do cliente.

Métodos: Cliente, getNome, getContato e getEndereco;

- Cliente: construtor da classe Cliente;
- getNome: retorna o nome do cliente;
- getContato: retorna o contato do cliente;
- getEndereco: retorna o endereço do cliente.

Classe Reserva

Atributos: idReserva, nomeCliente, quantidadePessoas, dataHora e status;

- idReserva: número inteiro com o ID da reserva;
- nomeCliente: *string* que armazena o nome do cliente que fez a reserva;
- quantidadePessoas: quantidade de pessoas que estão alocadas na reserva;
- dataHora: representa a data da reserva;
- status: *boolean* que informa se a reserva está ativa ou cancelada.

Métodos: Reserva, getIdReserva, getQuantidadePessoas, getStatus, getNomeCliente e getDataHora;

- Reserva: construtor da classe Reserva;

- getIdReserva: retorna o ID da reserva;
- getQuantidadePessoas: retorna a quantidade de pessoas da reserva;
- getStatus: retorna se a reserva esta ativa ou não;
- getNomeCliente: retorna o nome do cliente que fez a reserva;
- GetDataHora: retorna a data e horário da reserva.

Classe Cardápio

Atributos: itens;

- itens: vetor de *string*, com os pares de item e preço.

Métodos: Cardapio, adicionaItem, exibirCardapio e getCardapio;

- Cardapio: construtor da classe Cardapio;
- adicionaItem: Adiciona item e preço ao cardápio;
- exibirCardapio: exibe o cardápio;
- getCardapio: retorna o cardápio;

Classe Mesa

Atributos: status, quantidadeCadeiras e numeroMesa;

- status: *boolean* que informa se a mesa está disponível ou não;
- quantidadeCadeiras: número inteiro que informa a quantidade de cadeiras na mesa;
- numeroMesa: número da mesa.

Métodos: Mesa, getStatus, getQuantidadeCadeiras e getNumeroMesa;

- Mesa: construtor da classe Mesa;
- getStatus: retorna se a mesa está disponível ou não;
- getQuantidadeCadeiras: retorna a quantidade de cadeiras da mesa;
- getNumeroMesa: retorna o número da mesa.

3 Planejamento e Estrutura

O planejamento e a estrutura do sistema foram desenvolvidos com foco na modularidade, manutenção facilitada e adequação às situações reais enfrentadas por restaurantes. Para isso, foram aplicados conceitos fundamentais de programação orientada a objetos, como herança, composição e associação, conforme representado nos diagramas de classes e casos de uso.

No diagrama de classes, figura 2, a herança foi empregada para otimizar o *design* do sistema. Por exemplo, a superclasse base Pagamento define atributos genéricos, como o valor final, enquanto as classes-filhas (Pix, Dinheiro e Cartão) descrevem características específicas de cada método de pagamento. Essa abordagem promove a reutilização de atributos comuns, reduz a duplicidade de código e facilita a adição de novos métodos de pagamento no futuro.

De maneira semelhante, a classe base Funcionário foi utilizada para agrupar atributos e métodos comuns, como nome e matrícula, enquanto subclasses específicas representam os diferentes papéis no restaurante, como *Chef* de Cozinha, Caixa, Garçom e Recepcionista. Cada subclasse define características exclusivas que refletem suas responsabilidades no ambiente gastronômico. Por exemplo, o Recepcionista lida com reservas, enquanto o Garçom está associado ao atendimento das mesas.

Os relacionamentos no diagrama de classes foram projetados para refletir as interações reais entre os elementos do sistema. Um exemplo é o relacionamento entre Mesa e Garçom, que representa o fluxo dinâmico do atendimento de pedidos. Além disso, a interação entre Cliente, Reserva e Mesa foi detalhada para contemplar situações comuns no gerenciamento de restaurantes, como reservas para múltiplas mesas e a rotatividade das mesmas ao longo do tempo.

No diagrama de casos de uso, figura 1, os atores foram definidos para representar as principais interações com o sistema. Clientes podem realizar ações como visualizar o cardápio, fazer pedidos e reservas, enquanto os Funcionários, como Caixa e *Chef* de Cozinha, desempenham funções relacionadas ao preparo de comida, processamento de pagamentos e organização operacional. Essa abordagem garante que todas as funcionalidades essenciais estejam alinhadas com o fluxo de trabalho de um restaurante.

A estrutura foi planejada a fim de refletir situações reais de organizações em um restaurantes. As decisões de *design* orientadas aos objetos citadas visam desenvolver um sistema modular e escalável. Isso permite uma estrutura sólida para implementação e execução da simulação, que será abordada ao longo das seções 4 e 5, respectivamente.

4 Implementação e Funcionalidades

O diretório Trabalho é organizado da seguinte forma: *restaurant.cpp* (arquivo principal da simulação), Makefile e dois diretórios: *source* e *header*. A pasta *header* armazena arquivos de extensão *.hpp* que implementam as classes, pasta *source* armazena arquivos de extensão *.cpp* que implementam os métodos públicos da classe. A figura abaixo mostra a organização da pasta principal:

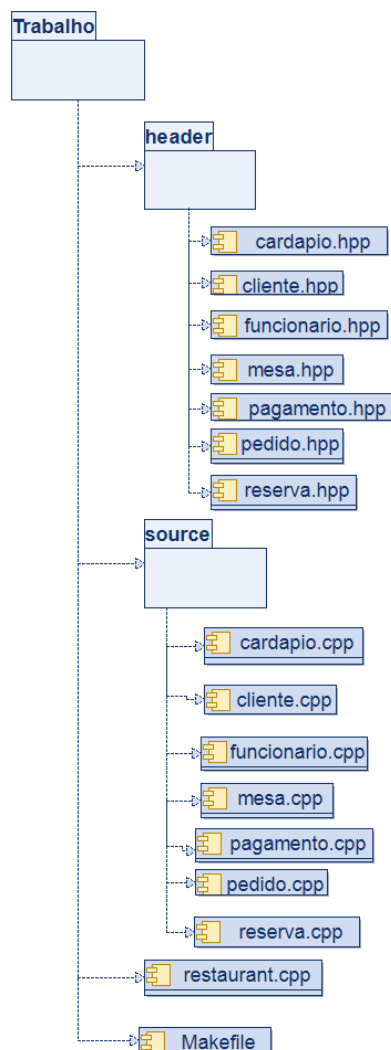


Figura 3: Organização do diretório.

O arquivo principal da simulação chama-se *restaurant.cpp*; na pasta *header* há os arquivos *cardapio.hpp*, *cliente.hpp*, *funcionario.hpp*, *mesa.hpp*, *pagamento.hpp*, *pedido.hpp* e *reserva.hpp* - eles implementam as classes de cardápio, cliente, funcionário, mesa, pagamento, pedido e reserva, respectivamente. Semelhantemente para a pasta *source* - nela há as implementações dos construtores, destrutores (se houver) e métodos de cada classe. O Makefile é um arquivo auxiliar para a compilação e execução do programa.

4.1 *restaurant.cpp*

O arquivo *restaurant.cpp* trata do arquivo principal do projeto. Nele há inclusão dos demais arquivos implementados para que a simulação possa ser executada.

O arquivo consta de uma função *displayDataHora* que exibe a data e horário no formato tradicional brasileiro: *dia/mês/ano hora:minuto*. Além disso, a função *main* contém as instanciações dos objetos para rodar a simulação.

4.2 Pasta *header*

4.2.1 *cardapio.hpp*

O arquivo *cardapio.hpp* trata da implementação da classe de Cardápio. A classe consta de um atributo privado: *itens*. Implementou-se um construtor para a classe.

4.2.2 *cliente.hpp*

O arquivo *cliente.hpp* trata da implementação da classe de Cliente. A classe consta de alguns atributos privados, como nome, contato e endereço do cliente. Implementou-se um construtor para a classe.

4.2.3 *funcionario.hpp*

O arquivo *funcionario.hpp* trata da implementação da classe abstrata Funcionário e as classes herdadas (Recepcionista, Garçom, *Chef* de Cozinha e Caixa). A classe abstrata possui atributos privados, como nome e matrícula - atributos comuns para qualquer tipo de funcionário. Além disso, há métodos públicos *get* e um método virtual, *realizarTarefa*, que torna a classe Funcionário abstrata. Implementou-se um construtor e um destrutor para a classe abstrata; classes herdadas possuem seus respectivos atributos, métodos e construtor.

4.2.4 *mesa.hpp*

O arquivo *mesa.hpp* trata da implementação da classe de Mesa. A classe consta de alguns atributos privados, como *status*, quantidade de cadeiras e número da mesa. Implementou-se um construtor para a classe.

4.2.5 *pagamento.hpp*

O arquivo *pagamento.hpp* trata da implementação da superclasse Pagamento e as classes herdadas (Dinheiro, Pix e Cartão). A superclasse possui atributos comuns a todas as classes. Implementou-se um construtor e um destrutor para a superclasse; classes herdadas possuem seus respectivos atributos, métodos e construtor.

4.2.6 *pedido.hpp*

O arquivo *pedido.hpp* trata da implementação da classe de Pedidos. A classe consta de atributos privados cliente (nome do cliente), mesa (número da mesa) e, semelhantemente ao atributo da classe Cardápio, *itens*. *Itens* é um vetor e em cada posição do vetor armazena-se um par: *string* com o nome do item e um número *integer*, que representa a quantidade pedida desse item. Implementou-se um construtor para a classe.

4.2.7 *reserva.hpp*

O arquivo *reserva.cpp* trata da implementação da classe Reserva. A classe consta de alguns atributos privados, como ID da reserva, quantidade de pessoas na reserva, *status* (ativa ou cancelada), nome do cliente da reserva, data e horário. Implementou-se um construtor para a classe.

4.3 Pasta *source*

4.3.1 *cardapio.cpp*

O arquivo *cardapio.cpp* trata da implementação dos métodos públicos da classe de Cardápio. A classe consta de métodos para adicionar itens, exibir o cardápio e retornar os itens de um cardápio.

4.3.2 *cliente.cpp*

O arquivo *cliente.cpp* trata da implementação dos métodos públicos da classe de Cliente. A classe consta de métodos *get* para retornar o nome, endereço e contato do cliente.

4.3.3 *funcionario.cpp*

O arquivo *funcionario.cpp* trata da implementação dos métodos públicos da classe de Funcionário. A classe consta de métodos *get* para retornar o nome e matrícula do funcionário. Além disso, cada classe herdada possui um método de realizar uma determinada tarefa, conforme o cargo do funcionário.

4.3.4 *mesa.cpp*

O arquivo *mesa.cpp* trata da implementação dos métodos públicos da classe de Mesa. A classe consta de métodos *get* para retornar o *status* da mesa, a quantidade de cadeiras e o número dela.

4.3.5 *pagamento.cpp*

O arquivo *pagamento.cpp* trata da implementação dos métodos públicos da classe de Pagamento. A classe consta de métodos para cálculos de valores totais e finais, incluindo ou não descontos. Há métodos *get* para um objeto Pedido e Cardápio para fazer o cálculo final de um determinado pedido, que deve estar em um cardápio. Além disso, cada classe herdada possui seus métodos, como pagamento em dinheiro, pix ou cartão, possíveis trocos, credores, confirmação e entre outros métodos.

4.3.6 *pedido.cpp*

O arquivo *pedido.cpp* trata da implementação dos métodos públicos da classe de Pedido. A classe consta de métodos *get* para retornar o pedido e o nome do cliente que o realizou, bem como o método *finalizaPedido* para limpar o vetor, quando encerra-se o pedido, após o pagamento dele, na simulação.

4.3.7 *reserva.cpp*

O arquivo *reserva.cpp* trata da implementação dos métodos públicos da classe de Reserva. A classe consta de métodos *get* para retornar o ID da reserva, a quantidade de pessoas, o *status*, o horário e nome do cliente que fez a reserva.

5 Guia de Uso e Exemplos

5.1 Guia de Uso

O projeto possui um arquivo Makefile, na pasta principal, com as regras para compilar o código, limpar e executar os arquivos de extensão *.o*.

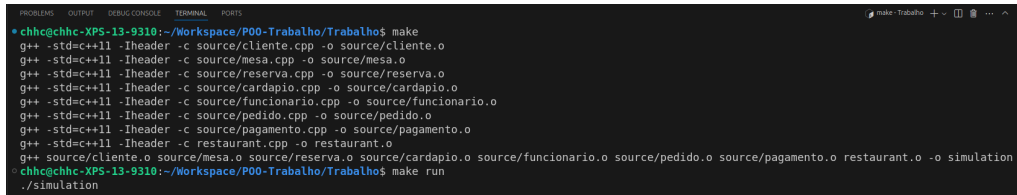
A pasta Trabalho, pasta principal, possui todos os arquivos necessários para compilar e executar a simulação. Após acessá-la, via terminal, basta digitar o comando *make* ou *make all* - isso irá compilar o código. A figura 4 mostra a maneira de compilar o código usando o comando *make*:



```
chhc@chhc-XPS-13-9310:~/Workspace/P00-Trabalho/Trabalho$ make
g++ -std=c++11 -Iheader -c source/cliente.cpp -o source/cliente.o
g++ -std=c++11 -Iheader -c source/mesa.cpp -o source/mesa.o
g++ -std=c++11 -Iheader -c source/reserva.cpp -o source/reserva.o
g++ -std=c++11 -Iheader -c source/cardapio.cpp -o source/cardapio.o
g++ -std=c++11 -Iheader -c source/funcionario.cpp -o source/funcionario.o
g++ -std=c++11 -Iheader -c source/pedido.cpp -o source/pedido.o
g++ -std=c++11 -Iheader -c source/pagamento.cpp -o source/pagamento.o
g++ -std=c++11 -Iheader -c restaurant.cpp -o restaurant.o
g++ source/cliente.o source/mesa.o source/reserva.o source/cardapio.o source/funcionario.o source/pedido.o source/pagamento.o restaurant.o -o simulation
chhc@chhc-XPS-13-9310:~/Workspace/P00-Trabalho/Trabalho$
```

Figura 4: Comando *make* para compilar a simulação.

Em seguida, digita-se, no terminal, o comando *make run* - isso executará a simulação. A figura 5 mostra a maneira de executar o código usando o comando *make run*:



```
chhc@chhc-XPS-13-9310:~/Workspace/P00-Trabalho/Trabalho$ make
g++ -std=c++11 -Iheader -c source/cliente.cpp -o source/cliente.o
g++ -std=c++11 -Iheader -c source/mesa.cpp -o source/mesa.o
g++ -std=c++11 -Iheader -c source/reserva.cpp -o source/reserva.o
g++ -std=c++11 -Iheader -c source/cardapio.cpp -o source/cardapio.o
g++ -std=c++11 -Iheader -c source/funcionario.cpp -o source/funcionario.o
g++ -std=c++11 -Iheader -c source/pedido.cpp -o source/pedido.o
g++ -std=c++11 -Iheader -c source/pagamento.cpp -o source/pagamento.o
g++ -std=c++11 -Iheader -c restaurant.cpp -o restaurant.o
g++ source/cliente.o source/mesa.o source/reserva.o source/cardapio.o source/funcionario.o source/pedido.o source/pagamento.o restaurant.o -o simulation
chhc@chhc-XPS-13-9310:~/Workspace/P00-Trabalho/Trabalho$ make run
./simulation
```

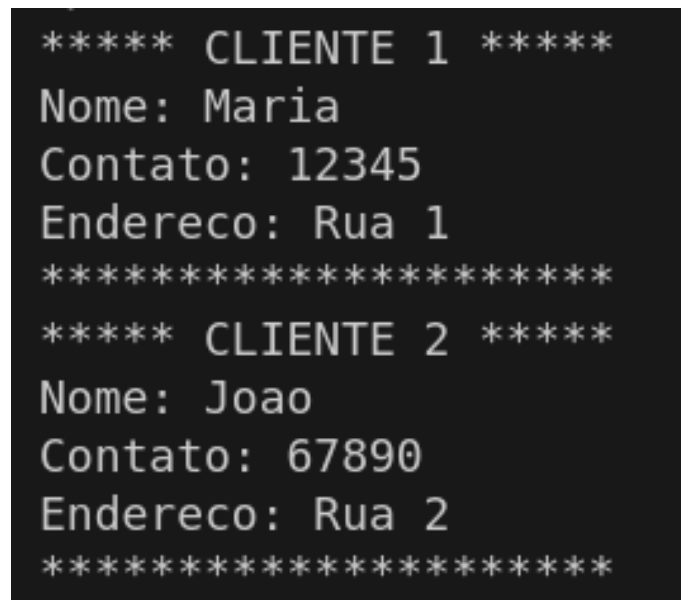
Figura 5: Comando *make run* para executar a simulação.

Assim, pode-se compilar e executar a simulação do restaurante. Além disso, desenvolveu-se o código com o compilador gcc (GCC) 13.2.0 em ambiente LINUX Ubuntu 24.04.1 LTS de sistema operacional.

5.2 Exemplos

Após executar os comandos *make* e *make all*, o código será executado. No terminal serão impressos os exemplos de objetos instanciados.

Há dois objetos da classe Cliente. Eles são mostrados na figura 6:



```
***** CLIENTE 1 *****
Nome: Maria
Contato: 12345
Endereco: Rua 1
*****
***** CLIENTE 2 *****
Nome: Joao
Contato: 67890
Endereco: Rua 2
*****
```

Figura 6: Exemplo da instanciação de dois objetos da classe Cliente.

Há dois objetos da classe Mesa. Eles são mostrados na figura 7:

```

***** MESA 1 *****
Status: Disponível
Quantidade de Cadeiras: 4
Numero da mesa: 1
*****

***** MESA 2 *****
Status: Indisponível
Quantidade de Cadeiras: 6
Numero da mesa: 2
*****

```

Figura 7: Exemplo da instanciação de dois objetos da classe Mesa.

Há um objeto da classe Reserva. Ele é mostrado na figura 8:

```

***** RESERVA 1 *****
ID da Reserva: 1
Nome do Cliente: Joao
Quantidade de Pessoas: 4
Status: Ativa
Data e Hora da Reserva: 10/11/2024 18:30
*****

```

Figura 8: Exemplo da instanciação de um objeto da classe Reserva.

Há um objeto da classe Cardápio. Ele é mostrado na figura 9:

```

*****
Cardápio:
Item: Pizza Margherita. Preço: R$ 25
Item: Hamburguer. Preço: R$ 18
Item: Suco de Laranja. Preço: R$ 8
*****

```

Figura 9: Exemplo da instanciação de um objeto da classe Cardápio.

Há quatro objetos da classe Funcionário: são eles uma recepcionista, um garçom, uma *chef* de cozinha e um caixa. Eles são mostrados na figura 10:

```

*** RECEPTIONISTA ***
Nome: Ana
Matricula: 101
Recepcionista Ana fala 3 idiomas.
*****
***** GARÇOM *****
Nome: Paulo
Matricula: 102
Garçom Paulo está servindo as mesas de Rota A.
*****
** CHEF DE COZINHA **
Nome: Beatriz
Matricula: 103
Chef de cozinha Beatriz tem especialidade nos pratos de Cozinha Italiana.
*****
***** CAIXA *****
Nome: Lucas
Matricula: 104
Caixa Lucas processa pagamentos no nível 5 de acesso.
*****

```

Figura 10: Exemplo da instanciação de objetos da classe Funcionário.

Há um objeto da classe Pedido. Ele é mostrado na figura 11:

```

*****
Cliente: Joao
Mesa: 1
Itens do Pedido:
- Pizza Margherita (Quantidade: 2)
- Suco de Laranja (Quantidade: 3)
*****

```

Figura 11: Exemplo da instanciação de um objeto da classe Pedido.

A classe Pagamento é interativa com o usuário que está rodando o código. Dessa forma fica sugerido ao leitor que execute o código e interaja para verificar o funcionamento de um pagamento ao restaurante, via dinheiro físico, pix ou cartão de crédito ou débito.

6 Desafios e Soluções

No decorrer do desenvolvimento do projeto o grupo esbarrou em alguns desafios, como a escolha do tema e a implementação de classes, métodos e heranças necessárias para que o sistema funcionasse de acordo com o que o grupo idealizou - ou prever e tratar possíveis erros que poderiam ser cometidos pelo usuário ao utilizar o sistema.

O primeiro desafio foi escolher o tema do projeto. Para isso, o grupo precisou pensar em algo que fosse possível aplicar os conceitos ensinados em aula e que pudesse cumprir tudo que foi requisitado pelo professor na descrição do trabalho. Então o grupo realizou uma reunião para discutir ideias e ao fim da reunião foi decidido que um sistema de gerenciamento de restaurante seria interessante de ser desenvolvido.

Após a escolha do tema, foi preciso pensar em todas as classes que deveriam ser implementadas nesse sistema, assim surgiu um segundo desafio: relacionar todas essas classes. Então, desenvolveu-se um caso de uso e um diagrama de classes para entender como esse sistema deveria funcionar estaticamente: listar os métodos e atributos de cada uma dessas classes e tornar mais fácil a visualização das relações entre todas as classes.

Feito isso, o próximo passo foi de fato programar e desenvolver o código com base em tudo que o grupo havia definido anteriormente. Um dos desafios que surgiram nessa etapa foi a divisão de responsabilidades para o desenvolvimento da simulação.

Por fim, depois de implementar todo o código, mesmo que funcionando corretamente se utilizado como o grupo esperava, ele ainda não estava pronto, pois o usuário podia interagir com o sistema de forma inadequada fazendo com que ele parasse de funcionar. Portanto, os integrantes do grupo trabalharam para identificar essas falhas no código e corrigi-las, fazendo com que o sistema ficasse mais robusto.

Um exemplo disso é o processo de pagamento. Essa foi uma parte do código desenvolvida para o usuário interagir com o sistema e realizar o pagamento do pedido de algum cliente. Em determinados momentos ele devia inserir valores

numéricos, mas nada o impede de inserir um caractere - nessa situação o sistema apresentava instabilidade. Portanto, para solucionar esse e demais problemas, foi necessário verificar todas possibilidades que poderiam tornar o sistema instável, como a inserção de valores inadequados.

6.1 Sugestões de Projeto

Com o trabalho finalizado, foram identificadas possíveis melhorias que poderiam ser feitas no sistema para torná-lo mais interessante e melhor para ser utilizado em um restaurante real.

A primeira sugestão seria tornar todos os processos interativos. Tendo em vista que o sistema desenvolvido pretendia simular o gerenciamento de um restaurante, toda a parte de reservar mesa, cadastrar um cliente, inserir um item no cardápio e até mesmo realizar um pedido não são realizados de forma interativa. Se o sistema fosse de fato implementado para ser utilizado em um restaurante real e não para simular o gerenciamento de um restaurante, todos esses processos teriam que ser feitos de forma interativa.

Além dessa, uma outra sugestão para melhorar o programa, seria desenvolver uma interface gráfica. Essa implementação traria uma grade evolução para o sistema, pois melhoraria a experiência de utilização, tornando-a mais intuitiva e amigável para o usuário.

A Implementação

Esta seção apresenta os códigos *header* e *source* da simulação de um restaurante.

Implementação do arquivo principal da simulação *restaurant.cpp*:

```
1  /*
2  SCC0604 - Programacao Orientada a Objetos
3  TRABALHO PRATICO FINAL
4
5  INTEGRANTES:
6      Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7      Henrique Carobolante Parro         NUSP: 11917987
8      Lucca Tommaso Monzani              NUSP: 5342324
9
10 Para compilar via terminal digite: make
11 Para executar via terminal digite: make run
12 Para limpar arquivos criados pelo Makefile via terminal digite: make clean
13
14 Arquivo principal da simulacao
15 */
16
17 #include "header/cardapio.hpp"
18 #include "header/cliente.hpp"
19 #include "header/funcionario.hpp"
20 #include "header/mesa.hpp"
21 #include "header/pagamento.hpp"
22 #include "header/pedido.hpp"
23 #include "header/reserva.hpp"
24
25 #include <iostream>
26 #include <iomanip>
27 #include <limits>
28
29 // Funcao auxiliar para exibir data e hora em formato tradicional: dia/mes/ano hora:minuto
30 void displayDataHora(const std::tm& dataHora)
31 {
32     std::cout << "Data e Hora da Reserva: " << std::put_time(&dataHora, "%d/%m/%Y %H:%M") << std::
33         endl;
34 }
35
36 int main()
37 {
38     //Instancia o objeto cliente1 e exibe informacoes
39     Cliente cliente1("Maria", "12345", "Rua 1");
40     std::cout << "***** CLIENTE 1 *****" << std::endl;
41     std::cout << "Nome: " << cliente1.getNome() << std::endl;
42     std::cout << "Contato: " << cliente1.getContato() << std::endl;
43     std::cout << "Endereco: " << cliente1.getEndereco() << std::endl;
44     std::cout << "*****" << std::endl;
45
46     //Instancia o objeto cliente2 e exibe informacoes
47     Cliente cliente2("Joao", "67890", "Rua 2");
48     std::cout << "***** CLIENTE 2 *****" << std::endl;
49     std::cout << "Nome: " << cliente2.getNome() << std::endl;
50     std::cout << "Contato: " << cliente2.getContato() << std::endl;
51     std::cout << "Endereco: " << cliente2.getEndereco() << std::endl;
52     std::cout << "*****" << std::endl;
53
54     std::cout << "\n";
55
56     //Instancia o objeto mesa1 e exibe informacoes
57     Mesa mesa1(true, 4, 1);
58     std::cout << "***** MESA 1 *****" << std::endl;
59     std::cout << "Status: " << (mesa1.getStatus() ? "Disponivel" : "Indisponivel") << std::endl;
```

```

59     std::cout << "Quantidade de Cadeiras: " << mesa1.getQuantidadeCadeiras() << std::endl;
60     std::cout << "Numero da mesa: " << mesa1.getNumeroMesa() << std::endl;
61     std::cout << "*****" << std::endl;
62
63     //Instancia o objeto mesa2 e exibe informacoes
64     Mesa mesa2(false, 6, 2);
65     std::cout << "***** MESA 2 *****" << std::endl;
66     std::cout << "Status: " << (mesa2.getStatus() ? "Disponivel" : "Indisponivel") << std::endl;
67     std::cout << "Quantidade de Cadeiras: " << mesa2.getQuantidadeCadeiras() << std::endl;
68     std::cout << "Numero da mesa: " << mesa2.getNumeroMesa() << std::endl;
69     std::cout << "*****" << std::endl;
70
71     std::cout << "\n";
72
73     // Define uma data e hora de exemplo para reserva
74     std::tm dataHora = {};
75     dataHora.tm_mday = 10;
76     dataHora.tm_mon = 11 - 1;
77     dataHora.tm_year = 2024 - 1900;
78     dataHora.tm_hour = 18;
79     dataHora.tm_min = 30;
80
81     // Instancia o objeto reserva1 e exibe detalhes da reserva
82     Reserva reserva1(1, 4, true, "Joao", dataHora);
83     std::cout << "***** RESERVA 1 *****" << std::endl;
84     std::cout << "ID da Reserva: " << reserva1.getIdReserva() << std::endl;
85     std::cout << "Nome do Cliente: " << reserva1.getNomeCliente() << std::endl;
86     std::cout << "Quantidade de Pessoas: " << reserva1.getQuantidadePessoas() << std::endl;
87     std::cout << "Status: " << (reserva1.getStatus() ? "Ativa" : "Inativa") << std::endl;
88     displayDataHora(dataHora);
89     std::cout << "*****" << std::endl;
90
91     std::cout << "\n";
92
93     // Instancia o objeto de cardapio, adiciona e exibe itens/precos
94     Cardapio cardapio;
95     std::cout << "*****" << std::endl;
96     cardapio.adicionaItem("Pizza Margherita", 25.0);
97     cardapio.adicionaItem("Hamburguer", 18.0);
98     cardapio.adicionaItem("Suco de Laranja", 8.0);
99     cardapio.exibirCardapio();
100    std::cout << "*****" << std::endl;
101
102    std::cout << "\n";
103
104    std::cout << "*** FUNCIONARIOS ***" << std::endl;
105    // Instancia objetos de funcionarios e exibe as tarefas
106    Recepcionista recepcionista("Ana", 101, 3);
107    std::cout << "*** RECEPCIONISTA ***" << std::endl;
108    std::cout << "Nome: " << recepcionista.getNome() << std::endl;
109    std::cout << "Matricula: " << recepcionista.getMatricula() << std::endl;
110    recepcionista.realizarTarefa();
111    std::cout << "*****" << std::endl;
112
113    Garcom garcom("Paulo", 102, "Rota A");
114    std::cout << "***** GAR OM *****" << std::endl;
115    std::cout << "Nome: " << garcom.getNome() << std::endl;
116    std::cout << "Matricula: " << garcom.getMatricula() << std::endl;
117    garcom.realizarTarefa();
118    std::cout << "*****" << std::endl;
119
120    ChefCozinha chefe("Beatriz", 103, "Cozinha Italiana");
121    std::cout << "** CHEF DE COZINHA **" << std::endl;
122    std::cout << "Nome: " << chefe.getNome() << std::endl;

```

```

123 std::cout << "Matricula: " << chefe.getMatricula() << std::endl;
124 chefe.realizarTarefa();
125 std::cout << "*****" << std::endl;
126
127 Caixa caixa("Lucas", 104, 5);
128 std::cout << "***** CAIXA *****" << std::endl;
129 std::cout << "Nome: " << caixa.getNome() << std::endl;
130 std::cout << "Matricula: " << caixa.getMatricula() << std::endl;
131 caixa.realizarTarefa();
132 std::cout << "*****" << std::endl;
133
134 std::cout << "\n";
135
136 // Instancia objeto de um pedido e exibe as informacoes
137 std::cout << "*****" << std::endl;
138 Pedido pedido1("Joao", 1);
139 pedido1.adicionalItem("Pizza Margherita", 2);
140 pedido1.adicionalItem("Suco de Laranja", 3);
141 pedido1.exibirPedido();
142 std::cout << "*****" << std::endl;
143
144 std::cout << "\n";
145
146 // Instancia objetos para pagamento e interage com o usuario para processar o pagamento do
147 // objeto pedido1
148 std::cout << "*****" << std::endl;
149 int formaPagamento, realizado = 0; // Valor para escolha da forma de pagaento e flag para
150 // indicar que o pagamento foi realizado
151
152 while(realizado == 0)
153 {
154     do
155     {
156         if (std::cin.fail())
157         {
158             std::cin.clear(); // Limpa o estado de erro
159             std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Descarta o
160             // restante da entrada inv lida
161             std::cout << "\nEntrada invalida. Insira um numero.\n" << std::endl;
162         }
163         std::cout << "Digite 1 para pagar com dinheiro. \nDigite 2 para pagar via Pix. \ndigite
164         // 3 para pagar com cartao."<<std::endl;
165         std::cout << "Digite aqui: ";
166         std::cin>>formaPagamento;
167     } while (std::cin.fail());
168
169     switch (formaPagamento)
170     {
171     case 1: // Opcao de
172             // pagamento em dinheiro
173             {
174                 Dinheiro pagamento1(&pedido1, cardapio); // Instancia
175                 // objeto de dinheiro
176                 pagamento1.pagarDinheiro();
177
178                 if(pagamento1.getConfirmacao() == 1) // Se
179                     // pagamento for bem sucedido
180                 {
181                     pagamento1.getPedido()->finalizaPedido();
182                     std::cout<<"Pagamento realizado com sucesso!"<<std::endl;
183                     realizado = 1;
184                 }
185                 if(pagamento1.getConfirmacao() == 2) // Se
186                     // pagamento for cancelado

```

```

179     {
180         std::cout<<"Pagamento cancelado!"<<std::endl;
181         realizado = 1;
182     }
183     if(pagamento1.getConfirmacao() != 1 && pagamento1.getConfirmacao() != 2)    //
184         Tratamento de erro
185     {
186         std::cout<<"Valor invalido, processo de pagamento reiniciado"<<std::endl;
187     }
188     break;
189 }
190 case 2:                                // Opcao de
191     pagamento via Pix
192 {
193     Pix pagamento1(&pedido1, cardapio);                                // Instancia
194     objeto com chave/credor de restaurante
195     pagamento1.setChave("restaurante@gmail.com");
196     pagamento1.setCredor("Restaurante");
197     pagamento1.pagarPix();
198
199     if(pagamento1.getConfirmacao() == 1)                                // Se
200         pagamento for bem sucedido
201     {
202         pagamento1.getPedido()->finalizaPedido();
203         std::cout<<"Pagamento realizado com sucesso!"<<std::endl;
204         realizado = 1;
205     }
206     if(pagamento1.getConfirmacao() == 2)                                // Se
207         pagamento for cancelado
208     {
209         std::cout<<"Pagamento cancelado!"<<std::endl;
210         realizado = 1;
211     }
212     if(pagamento1.getConfirmacao() != 1 && pagamento1.getConfirmacao() != 2)    //
213         Tratamento de erro
214     {
215         std::cout<<"Valor invalido, processo de pagamento reiniciado"<<std::endl;
216     }
217     break;
218 }
219 case 3:                                // Opcao de
220     pagamento via cartao
221 {
222     Cartao pagamento1(&pedido1, cardapio);                                // Instancia
223     objeto de cartao
224     pagamento1.pagarCartao();
225     if(pagamento1.getConfirmacao() == 1)                                // Se
226         pagamento for bem sucedido
227     {
228         pagamento1.getPedido()->finalizaPedido();
229         std::cout<<"Pagamento realizado com sucesso!"<<std::endl;
230         realizado = 1;
231     }
232     if(pagamento1.getConfirmacao() == 2)                                // Se
233         pagamento for cancelado
234     {
235         std::cout<<"Pagamento cancelado!"<<std::endl;
236         realizado = 1;
237     }
238     if(pagamento1.getConfirmacao() != 1 && pagamento1.getConfirmacao() != 2)    //
239         Tratamento de erro
240     {
241         std::cout<<"Valor invalido, processo de pagamento reiniciado"<<std::endl;
242     }
243 }

```

```

232         break;
233     }
234     default:
235         break;
236     }
237 }
238
239 std::cout << "*****" << std::endl;
240
241 return 0;
242 }

```

Código 1: Arquivo *restaurant.cpp*.

Implementação dos arquivos *cardapio.hpp* e *cardapio.cpp*, respectivamente:

```

1  /*
2  SCC0604 - Programacao Orientada a Objetos
3  TRABALHO PRATICO FINAL
4
5  INTEGRANTES:
6      Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7      Henrique Carobolante Parro          NUSP: 11917987
8      Lucca Tommaso Monzani               NUSP: 5342324
9
10     Arquivo header com a declaracao da classe de cardapio
11  */
12
13  #ifndef CARDAPIO_HPP
14  #define CARDAPIO_HPP
15
16  #include <string>
17  #include <vector>
18  #include <utility> // Para usar o std::pair
19
20  class Cardapio
21  {
22  private:
23      std::vector<std::pair<std::string, double>> _itens;
24
25  public:
26      // Construtor da classe
27      Cardapio();
28
29      // Metodos da classe
30      void adicionaItem(const std::string& item, double preco);
31      void exibirCardapio() const;
32      std::vector<std::pair<std::string, double>> getCardapio();
33  };
34
35  #endif // CARDAPIO_HPP

```

Código 2: Arquivo *cardapio.hpp*.

```

1  /*
2  SCC0604 - Programacao Orientada a Objetos
3  TRABALHO PRATICO FINAL
4
5  INTEGRANTES:
6      Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7      Henrique Carobolante Parro          NUSP: 11917987
8      Lucca Tommaso Monzani               NUSP: 5342324
9
10     Arquivo source com a implementacao dos metodos publicos da classe de cardapio

```

```

11     Link de estudo sobre biblioteca utility e funcao pair: https://terminalroot.com.br/2021/08/cpp-
        pair-e-tuple.html
12 */
13
14 #include "cardapio.hpp"
15 #include <iostream>
16
17 // Construtor
18 Cardapio::Cardapio() {}
19
20 void Cardapio::adicionaItem(const std::string& item, double preco) // Adiciona item e preco ao
        cardapio
21 {
22     _itens.push_back(std::make_pair(item, preco));
23 }
24
25 void Cardapio::exibirCardapio() const // Exibe o cardapio
26 {
27     std::cout << "Card pio:\n";
28     for (const auto& par : _itens)
29         std::cout << "Item: " << par.first << ". Pre o: R$ " << par.second << std::endl;
30 }
31
32 std::vector<std::pair<std::string, double>> Cardapio::getCardapio() // Retorna o cardapio
33 {
34     return _itens;
35 }

```

Código 3: Arquivo *cardapio.cpp*.

Implementação dos arquivos *cliente.hpp* e *cliente.cpp*, respectivamente:

```

1  /*
2  SCC0604 - Programacao Orientada a Objetos
3  TRABALHO PRATICO FINAL
4
5  INTEGRANTES:
6      Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7      Henrique Carobolante Parro          NUSP: 11917987
8      Lucca Tommaso Monzani               NUSP: 5342324
9
10     Arquivo header com a declaracao da classe de cliente
11 */
12
13 #ifndef CLIENTE_HPP
14 #define CLIENTE_HPP
15
16 #include <string>
17
18 class Cliente
19 {
20     private:
21         std::string _nome;
22         std::string _contato;
23         std::string _endereco;
24
25     public:
26         // Construtor da classe
27         Cliente(std::string nome, std::string contato, std::string endereco);
28
29         // Metodos da classe
30         std::string getNome() const;
31         std::string getContato() const;
32         std::string getEndereco() const;
33 };

```

```

34
35 #endif // CLIENTE_HPP

```

Código 4: Arquivo *cliente.hpp*.

```

1  /*
2   SCC0604 - Programacao Orientada a Objetos
3   TRABALHO PRATICO FINAL
4
5   INTEGRANTES:
6       Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7       Henrique Carobolante Parro          NUSP: 11917987
8       Lucca Tommaso Monzani               NUSP: 5342324
9
10  Arquivo source com a implementacao dos metodos publicos da classe de cliente
11  */
12
13  #include "../header/cliente.hpp"
14
15  // Construtor
16  Cliente::Cliente(std::string nome, std::string contato, std::string endereco): _nome(nome), _contato
    (contato), _endereco(endereco) {}
17
18  std::string Cliente::getNome() const          // Retorna o nome do cliente
19  {
20      return _nome;
21  }
22
23  std::string Cliente::getContato() const        // Retorna o contato do cliente
24  {
25      return _contato;
26  }
27
28  std::string Cliente::getEndereco() const      // Retorna o endereco do cliente
29  {
30      return _endereco;
31  }

```

Código 5: Arquivo *cliente.cpp*.

Implementação dos arquivos *funcionario.hpp* e *funcionario.cpp*, respectivamente:

```

1  /*
2   SCC0604 - Programacao Orientada a Objetos
3   TRABALHO PRATICO FINAL
4
5   INTEGRANTES:
6       Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7       Henrique Carobolante Parro          NUSP: 11917987
8       Lucca Tommaso Monzani               NUSP: 5342324
9
10  Arquivo header com a declaracao da classe de funcionario
11  */
12
13  #ifndef FUNCIONARIO_HPP
14  #define FUNCIONARIO_HPP
15
16  #include <iostream>
17  #include <string>
18
19  class Funcionario                          // Classe abstrata Funcionario
20  {
21      private:
22          std::string _nome;
23          int _matricula;

```



```

24
25     public:
26         // Construtor e destrutor da classe
27         Funcionario(std::string nome, int matricula);
28         virtual ~Funcionario() = default;
29
30         // Metodos da classe
31         virtual void realizarTarefa() const = 0;
32         std::string getNome() const;
33         int getMatricula() const;
34 };
35
36 class Recepcionista : public Funcionario // Classe herdada de funcionario
37 {
38     private:
39         int _idiomas;
40
41     public:
42         // Construtor da classe
43         Recepcionista(std::string nome, int matricula, int idiomas);
44
45         // Metodo da classe
46         void realizarTarefa() const override;
47 };
48
49 class Garcom : public Funcionario // Classe herdada de funcionario
50 {
51     private:
52         std::string _rotaAtendimento;
53
54     public:
55         // Construtor da classe
56         Garcom(std::string nome, int matricula, std::string rotaAtendimento);
57
58         // Metodo da classe
59         void realizarTarefa() const override;
60 };
61
62 class ChefCozinha : public Funcionario // Classe herdada de funcionario
63 {
64     private:
65         std::string _especialidade;
66
67     public:
68         // Construtor da classe
69         ChefCozinha(std::string nome, int matricula, std::string especialidade);
70
71         // Metodo da classe
72         void realizarTarefa() const override;
73 };
74
75 class Caixa : public Funcionario // Classe herdada de funcionario
76 {
77     private:
78         int _nivelAcesso;
79
80     public:
81         // Construtor da classe
82         Caixa(std::string nome, int matricula, int nivelAcesso);
83
84         // Metodo da classe
85         void realizarTarefa() const override;
86 };
87

```

```
88 #endif // FUNCIONARIO_HPP
```

Código 6: Arquivo *funcionario.hpp*.

```
1  /*
2  SCC0604 - Programacao Orientada a Objetos
3  TRABALHO PRATICO FINAL
4
5  INTEGRANTES:
6      Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7      Henrique Carobolante Parro          NUSP: 11917987
8      Lucca Tommaso Monzani               NUSP: 5342324
9
10     Arquivo source com a implementacao dos metodos publicos da classe de funcionario
11 */
12
13 #include "../header/funcionario.hpp"
14
15 // Construtor da classe Funcionario
16 Funcionario::Funcionario(std::string nome, int matricula): _nome(nome), _matricula(matricula) {}
17
18 std::string Funcionario::getNome() const           // Retorna o nome do funcionario
19 {
20     return _nome;
21 }
22
23 int Funcionario::getMatricula() const              // Retorna a matricula do funcionario
24 {
25     return _matricula;
26 }
27
28 // Construtor da classe Recepcionista
29 Recepcionista::Recepcionista(std::string nome, int matricula, int idiomas): Funcionario(nome,
30     matricula), _idiomas(idiomas) {}
31
32 void Recepcionista::realizarTarefa() const         // Retorna a atividade de recepcionista
33 {
34     std::cout << "Recepcionista " << getNome() << " fala " << _idiomas << " idiomas.\n";
35 }
36
37 // Construtor da classe Garcom
38 Garcom::Garcom(std::string nome, int matricula, std::string rotaAtendimento): Funcionario(nome,
39     matricula), _rotaAtendimento(rotaAtendimento) {}
40
41 void Garcom::realizarTarefa() const                // Retorna a atividade de garcom
42 {
43     std::cout << "Gar om " << getNome() << " est servindo as mesas de " << _rotaAtendimento << "
44         .\n";
45 }
46
47 // Construtor da classe ChefCozinha
48 ChefCozinha::ChefCozinha(std::string nome, int matricula, std::string especialidade): Funcionario(
49     nome, matricula), _especialidade(especialidade) {}
50
51 void ChefCozinha::realizarTarefa() const           // Retorna a atividade de chef de cozinha
52 {
53     std::cout << "Chef de cozinha " << getNome() << " tem especialidade nos pratos de " <<
54         _especialidade << ".\n";
55 }
56
57 // Construtor da classe Caixa
58 Caixa::Caixa(std::string nome, int matricula, int nivelAcesso): Funcionario(nome, matricula),
59     _nivelAcesso(nivelAcesso) {}
60
```

```

55 void Caixa::realizarTarefa() const           // Retorna a atividade de caixa
56 {
57     std::cout << "Caixa " << getNome() << " processa pagamentos no n vel " << _nivelAcesso << " de
        acesso.\n";
58 }

```

Código 7: Arquivo *funcionario.cpp*.

Implementação dos arquivos *mesa.hpp* e *mesa.cpp*, respectivamente:

```

1  /*
2  SCC0604 - Programacao Orientada a Objetos
3  TRABALHO PRATICO FINAL
4
5  INTEGRANTES:
6      Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7      Henrique Carobolante Parro          NUSP: 11917987
8      Lucca Tommaso Monzani               NUSP: 5342324
9
10     Arquivo header com a declaracao da classe de mesa
11 */
12
13 #ifndef MESA_HPP
14 #define MESA_HPP
15
16 class Mesa
17 {
18     private:
19         bool _status;           // True se mesa disponivel; false se mesa indisponivel
20         int _quantidadeCadeiras;
21         int _numeroMesa;
22
23     public:
24         // Construtor da classe
25         Mesa(bool status, int quantidadeCadeiras, int numeroMesa);
26
27         // Metodos da classe
28         bool getStatus() const;
29         int getQuantidadeCadeiras() const;
30         int getNumeroMesa() const;
31 };
32
33 #endif // MESA_HPP

```

Código 8: Arquivo *mesa.hpp*.

```

1  /*
2  SCC0604 - Programacao Orientada a Objetos
3  TRABALHO PRATICO FINAL
4
5  INTEGRANTES:
6      Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7      Henrique Carobolante Parro          NUSP: 11917987
8      Lucca Tommaso Monzani               NUSP: 5342324
9
10     Arquivo source com a implementacao dos metodos publicos da classe de mesa
11 */
12
13 #include "mesa.hpp"
14
15 // Construtor da classe
16 Mesa::Mesa(bool status, int quantidadeCadeiras, int numeroMesa): _status(status),
    _quantidadeCadeiras(quantidadeCadeiras), _numeroMesa(numeroMesa) {}
17
18 bool Mesa::getStatus() const           // Retorna se a mesa esta disponivel ou nao

```

```

19 {
20     return _status;
21 }
22
23 int Mesa::getQuantidadeCadeiras() const // Retorna a quantidade de cadeiras da mesa
24 {
25     return _quantidadeCadeiras;
26 }
27
28 int Mesa::getNumeroMesa() const // Retorna o numero da mesa
29 {
30     return _numeroMesa;
31 }

```

Código 9: Arquivo *mesa.cpp*.

Implementação dos arquivos *pagamento.hpp* e *pagamento.cpp*, respectivamente:

```

1  /*
2   SCC0604 - Programacao Orientada a Objetos
3   TRABALHO PRATICO FINAL
4
5   INTEGRANTES:
6       Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7       Henrique Carobolante Parro         NUSP: 11917987
8       Lucca Tommaso Monzani              NUSP: 5342324
9
10  Arquivo header com a declaracao da classe de pagamento
11  */
12
13 #ifndef PAGAMENTO_HPP
14 #define PAGAMENTO_HPP
15
16 #include "pedido.hpp"
17 #include "cardapio.hpp"
18
19 #include <string>
20 #include <vector>
21
22 class Pagamento // Superclasse
23 {
24     private:
25         double valorTotal;
26         double valorDesconto;
27         double valorFinal;
28         int erro;
29         Pedido* p; // Ponteiro para Pedido
30         Cardapio c;
31
32     public:
33         // Construtor e destrutor da classe
34         Pagamento(Pedido* p_in, Cardapio c_in, float total = 0, int erro = 0);
35         virtual ~Pagamento() = default;
36
37         // Metodos da classe
38         void calcTotal();
39         void setDesconto();
40         void semDesconto();
41         void setFinal();
42         double getFinal() const;
43         Pedido* getPedido() const;
44         Cardapio getCardapio() const;
45         int getErro() const;
46         float getTotal() const;
47 };

```

```

48
49 class Dinheiro : public Pagamento          // Classe herdada de pagamento
50 {
51     private:
52         int confirmacao;
53         double valorRecebido;
54         double troco;
55
56     public:
57         // Construtor da classe
58         Dinheiro(Pedido* p_in, Cardapio c_in, float total = 0, int erro = 0, int conf = 0);
59
60         // Metodos da classe
61         void pagarDinheiro();
62         int getConfirmacao() const;
63         double getTroco() const;
64         double getValor() const;
65 };
66
67 class Pix : public Pagamento                // Classe herdada de pagamento
68 {
69     private:
70         int confirmacao;
71         std::string devedor;
72         std::string credor;
73         std::string chave;
74
75     public:
76         // Construtor da classe
77         Pix(Pedido* p_in, Cardapio c_in, float total = 0, int erro = 0);
78
79         // Metodos da classe
80         void pagarPix();
81         int getConfirmacao() const;
82         void setCredor(const std::string& cred);
83         void setChave(const std::string& ch);
84 };
85
86 class Cartao : public Pagamento             // Classe herdada de pagamento
87 {
88     private:
89         int confirmacao;
90         int tipoPagamento;
91         std::string banco;
92
93     public:
94         // Construtor da classe
95         Cartao(Pedido* p_in, Cardapio c_in, float total = 0, int erro = 0);
96
97         // Metodos da classe
98         void pagarCartao();
99         int getConfirmacao() const;
100        int getTipo() const;
101 };
102
103 #endif // PAGAMENTO_HPP

```

Código 10: Arquivo *pagamento.hpp*.

```

1  /*
2  SCC0604 - Programacao Orientada a Objetos
3  TRABALHO PRATICO FINAL
4
5  INTEGRANTES:

```

```

6      Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7      Henrique Carobolante Parro         NUSP: 11917987
8      Lucca Tommaso Monzani              NUSP: 5342324
9
10     Arquivo source com a implementacao dos metodos publicos da classe de pagamento
11 */
12
13 #include "../header/pagamento.hpp"
14
15 #include <iostream>
16 #include <iomanip>
17 #include <limits>
18
19 // Construtor da classe Pagamento
20 Pagamento::Pagamento(Pedido *p_in, Cardapio c_in, float total, int erro) : valorTotal(total), erro(
    erro), p(p_in), c(c_in) {}
21
22 void Pagamento::calcTotal() // Calcula o valor total da conta
23 {
24     auto aux_pedido = p->getPedido();
25     auto aux_cardapio = c.getCardapio();
26
27     for (const auto &par_pedido : aux_pedido)
28     {
29         bool encontrado = false;
30
31         for (const auto &par_cardapio : aux_cardapio)
32         {
33             if (par_pedido.first == par_cardapio.first)
34             {
35                 encontrado = true;
36                 valorTotal += par_pedido.second * par_cardapio.second;
37                 break;
38             }
39         }
40
41         if (!encontrado) // Tratamento para itens nao encontrados
42         {
43             std::cout << "0 item " << par_pedido.first << " nao foi encontrado no cardapio." << std
                ::endl;
44             erro = 1;
45         }
46     }
47 }
48
49 void Pagamento::setDesconto() // Adiciona desconto ao preco
50 {
51     do
52     {
53         do
54         {
55             if (std::cin.fail())
56             {
57                 std::cin.clear(); // Limpa o estado de erro
58                 std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Descarta o
                    restante da entrada inv lida
59                 std::cout << "\nEntrada invalida. Insira um numero.\n" << std::endl;
60             }
61             std::cout << "\nInsira o desconto a ser aplicado: R$ ";
62             std::cin >> valorDesconto;
63         } while (std::cin.fail());
64
65         if (valorDesconto > getTotal())
66             std::cout << "\nDesconto inserido invalido!\nValor maior que o total do pedido!" << std

```

```

        ::endl;
67     } while (valorDesconto > getTotal());
68 }
69
70 void Pagamento::semDesconto() // Pagamento final sem desconto
71 {
72     valorFinal = valorTotal;
73 }
74
75 void Pagamento::setFinal() // Pagamento final com desconto
76 {
77     valorFinal = valorTotal - valorDesconto;
78 }
79
80 double Pagamento::getFinal() const // Retorna o valor final
81 {
82     return valorFinal;
83 }
84
85 Pedido *Pagamento::getPedido() const // Retorna um ponteiro para o pedido
86 {
87     return p;
88 }
89
90 Cardapio Pagamento::getCardapio() const // Retorna a classe do cardapio
91 {
92     return c;
93 }
94
95 int Pagamento::getErro() const
96 {
97     return erro;
98 }
99
100 float Pagamento::getTotal() const // Calcula o valor final
101 {
102     return valorTotal;
103 }
104
105 // Construtor da classe Dinheiro
106 Dinheiro::Dinheiro(Pedido *p_in, Cardapio c_in, float total, int erro, int conf) : Pagamento(p_in,
    c_in, total, erro), confirmacao(conf) {}
107
108 void Dinheiro::pagarDinheiro() // Metodo interativo para realizar pagamento
    em dinheiro
109 {
110     valorRecebido = 0;
111     int realizado = 0;
112     calcTotal();
113
114     std::cout << "Pagamento em dinheiro selecionado." << std::endl;
115     std::cout << "Total a ser pago: R$ " << getTotal() << std::endl;
116
117     while (realizado == 0) // Enquanto o pagamento nao for realizado
118     {
119         do
120         {
121             if (std::cin.fail())
122             {
123                 std::cin.clear(); // Limpa o estado de erro
124                 std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Descarta o
                    restante da entrada inv lida
125                 std::cout << "\nEntrada invalida. Insira um numero.\n" << std::endl;
126             }

```

```

127     std::cout << "Digite 1 para aplicar desconto.\n" << "Digite 2 para continuar sem
        desconto." << std::endl;
128     std::cout << "Digite aqui: ";
129     std::cin >> confirmacao;
130 } while (std::cin.fail());
131
132 if (confirmacao == 1) // Se houver desconto
133 {
134     setDesconto();
135     setFinal();
136     std::cout << "Valor final a ser pago: R$ " << getFinal() << std::endl;
137 }
138 if (confirmacao == 2) // Se nao houver desconto
139 {
140     semDesconto();
141     std::cout << "Valor final a ser pago: R$ " << getTotal() << std::endl;
142 }
143 if (confirmacao == 1 || confirmacao == 2)
144     realizado = 1;
145 else
146     std::cout << "Valor invalido!" << std::endl;
147 }
148
149 while (valorRecebido < getFinal()) // Processa pagamento do cliente, desde que
    seja maior ou igual ao valor final
150 {
151     do
152     {
153         if (std::cin.fail())
154         {
155             std::cin.clear(); // Limpa o estado de erro
156             std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Descarta o
                restante da entrada inv lida
157             std::cout << "\nEntrada invalida. Insira um numero.\n" << std::endl;
158         }
159         std::cout << "Insira a quantia paga pelo cliente: R$ ";
160         std::cin >> valorRecebido;
161     } while (std::cin.fail());
162
163     if (valorRecebido < getFinal())
164         std::cout << "Valor insuficiente!" << std::endl;
165 }
166
167 // Calculo de troco
168 troco = valorRecebido - getFinal();
169 std::cout << "Troco calculado: R$" << troco << std::endl;
170
171 while (realizado == 1)
172 {
173     do
174     {
175         if (std::cin.fail())
176         {
177             std::cin.clear(); // Limpa o estado de erro
178             std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Descarta o
                restante da entrada inv lida
179             std::cout << "\nEntrada invalida. Insira um numero.\n" << std::endl;
180         }
181         std::cout << "Digite 1 para confirmar pagamento.\n" << "Digite 2 para cancelar processo
                de pagamento." << std::endl;
182         std::cout << "Digite aqui: ";
183         std::cin >> confirmacao;
184     } while (std::cin.fail());
185

```



```

186         if (confirmacao == 1 || confirmacao == 2)
187             realizado = 2;
188         else
189             std::cout << "Valor invalido!" << std::endl;
190     }
191 }
192
193 int Dinheiro::getConfirmacao() const // Retorna status da confirmacao
194 {
195     return confirmacao;
196 }
197
198 double Dinheiro::getTroco() const // Retorna o valor do troco
199 {
200     return troco;
201 }
202
203 double Dinheiro::getValor() const // Retorna o valor recebido pelo cliente
204 {
205     return valorRecebido;
206 }
207
208 // Construtor da classe Pix
209 Pix::Pix(Pedido *p_in, Cardapio c_in, float total, int erro) : Pagamento(p_in, c_in, total, erro) {}
210
211 void Pix::pagarPix() // Metodo interativo para realizar pagamento via
212     Pix
213 {
214     int realizado = 0;
215     calcTotal();
216
217     std::cout << "Pagamento via Pix selecionado." << std::endl;
218     std::cout << "Total a ser pago: R$ " << getTotal() << std::endl;
219
220     while (realizado == 0) // Enquanto o pagamento nao for realizado
221     {
222         do
223         {
224             if (std::cin.fail())
225             {
226                 std::cin.clear(); // Limpa o estado de erro
227                 std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Descarta o
228                     restante da entrada inv lida
229                 std::cout << "\nEntrada invalida. Insira um numero.\n" << std::endl;
230             }
231             std::cout << "Digite 1 para aplicar desconto.\n" << "Digite 2 para continuar sem
232                 desconto." << std::endl;
233             std::cout << "Digite aqui: ";
234             std::cin >> confirmacao;
235             } while (std::cin.fail());
236
237             if (confirmacao == 1) // Se houver desconto
238             {
239                 setDesconto();
240                 setFinal();
241                 std::cout << "Valor final a ser pago: R$ " << getFinal() << std::endl;
242             }
243             if (confirmacao == 2) // Se nao houver desconto
244             {
245                 semDesconto();
246                 std::cout << "Valor final a ser pago: R$ " << getFinal() << std::endl;
247             }
248             if (confirmacao == 1 || confirmacao == 2)
249                 realizado = 1;

```

```

247     else
248         std::cout << "Valor invalido!" << std::endl;
249     }
250
251     devedor = getPedido()->getCliente();
252
253     std::cout << "Devedor: " << devedor << std::endl;
254     std::cout << "Credor: " << credor << std::endl;
255     std::cout << "Chave Pix: " << chave << std::endl;
256
257     while (realizado == 1)
258     {
259         do
260         {
261             if (std::cin.fail())
262             {
263                 std::cin.clear(); // Limpa o estado de erro
264                 std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Descarta o
265                     restante da entrada inv lida
266                 std::cout << "\nEntrada invalida. Insira um numero.\n" << std::endl;
267             }
268             std::cout << "Digite 1 para confirmar pagamento.\n" << "Digite 2 para cancelar processo
269                 de pagamento." << std::endl;
270             std::cout << "Digite aqui: ";
271             std::cin >> confirmacao;
272             } while (std::cin.fail());
273
274             if (confirmacao == 1 || confirmacao == 2)
275                 realizado = 2;
276             else
277                 std::cout << "Valor invalido!" << std::endl;
278         }
279     }
280
281     int Pix::getConfirmacao() const // Retorna status da confirmacao
282     {
283         return confirmacao;
284     }
285
286     void Pix::setCredor(const std::string &cred) // Retorna o credor do Pix
287     {
288         credor = cred;
289     }
290
291     void Pix::setChave(const std::string &ch) // Retorna a chave Pix do credor
292     {
293         chave = ch;
294     }
295
296     // Construtor da classe Cartao
297     Cartao::Cartao(Pedido *p_in, Cardapio c_in, float total, int erro) : Pagamento(p_in, c_in, total,
298         erro) {}
299
300     void Cartao::pagarCartao() // Metodo interativo para realizar pagamento com
301         cartao
302     {
303         calcTotal();
304         int realizado = 0;
305
306         std::cout << "Pagamento via cartao selecionado." << std::endl;
307         std::cout << "Total a ser pago: R$ " << getTotal() << std::endl;
308
309         while (realizado == 0) // Enquanto o pagamento nao for realizado
310         {

```

```

307     do
308     {
309         if (std::cin.fail())
310         {
311             std::cin.clear(); // Limpa o estado de erro
312             std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Descarta o
313                 restante da entrada inv lida
314             std::cout << "\nEntrada invalida. Insira um numero.\n" << std::endl;
315         }
316         std::cout << "Digite 1 para aplicar desconto.\n" << "Digite 2 para continuar sem
317             desconto." << std::endl;
318         std::cout << "Digite aqui: ";
319         std::cin >> confirmacao;
320     } while (std::cin.fail());
321
322     if (confirmacao == 1) // Se houver desconto
323     {
324         setDesconto();
325         setFinal();
326         std::cout << "Valor final a ser pago: R$ " << getFinal() << std::endl;
327     }
328     if (confirmacao == 2) // Se nao houver desconto
329     {
330         semDesconto();
331         std::cout << "Valor final a ser pago: R$ " << getFinal() << std::endl;
332     }
333
334     if (confirmacao == 1 || confirmacao == 2)
335         realizado = 1;
336     else
337         std::cout << "Valor invalido!" << std::endl;
338 }
339
340 do
341 {
342     if (std::cin.fail())
343     {
344         std::cin.clear(); // Limpa o estado de erro
345         std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Descarta o
346             restante da entrada inv lida
347         std::cout << "\nEntrada invalida. Insira um numero.\n" << std::endl;
348     }
349     std::cout << "Digite o nome do banco: ";
350     std::getline(std::cin >> std::ws, banco);
351 } while (std::cin.fail());
352
353 while (realizado == 1)
354 {
355     do
356     {
357         if (std::cin.fail())
358         {
359             std::cin.clear(); // Limpa o estado de erro
360             std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Descarta o
361                 restante da entrada inv lida
362             std::cout << "\nEntrada invalida. Insira um numero.\n" << std::endl;
363         }
364         std::cout << "\n";
365
366         std::cout << "Digite 1 para pagamento no credito.\nDigite 2 para pagamento no debito."
367             << std::endl;
368         std::cout << "Digite aqui: ";
369         std::cin >> tipoPagamento;
370     } while (std::cin.fail());

```

```

366
367     if (tipoPagamento == 1 || tipoPagamento == 2)
368         realizado = 2;
369     else
370         std::cout << "Valor invalido!" << std::endl;
371 }
372 if (tipoPagamento == 1) // Seleciona a opcao de pagamento no credito
373     std::cout << "Credito selecionado.\n" << "Banco: " << banco << std::endl;
374
375 if (tipoPagamento == 2) // Seleciona a opcao de pagamento no debito
376     std::cout << "Debito selecionado.\n" << "Banco: " << banco << std::endl;
377
378 while (realizado == 2)
379 {
380     do
381     {
382         if (std::cin.fail())
383         {
384             std::cin.clear(); // Limpa o estado de erro
385             std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Descarta o
386                 restante da entrada inv lida
387             std::cout << "\nEntrada invalida. Insira um numero.\n" << std::endl;
388         }
389         std::cout << "Digite 1 para confirmar pagamento.\n" << "Digite 2 para cancelar processo
390             de pagamento." << std::endl;
391         std::cout << "Digite aqui: ";
392         std::cin >> confirmacao;
393     } while (std::cin.fail());
394
395     if (confirmacao == 1 || confirmacao == 2)
396     {
397         realizado = 3;
398     }
399     else
400     {
401         std::cout << "Valor invalido!" << std::endl;
402     }
403 }
404
405 int Cartao::getConfirmacao() const // Retorna status da confirmacao
406 {
407     return confirmacao;
408 }
409
410 int Cartao::getTipo() const // Retorna a opcao do tipo de pagamento
411 {
412     return tipoPagamento;
413 }

```

Código 11: Arquivo *pagamento.cpp*.

Implementação dos arquivos *pedido.hpp* e *pedido.cpp*, respectivamente:

```

1  /*
2  SCC0604 - Programacao Orientada a Objetos
3  TRABALHO PRATICO FINAL
4
5  INTEGRANTES:
6      Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7      Henrique Carobolante Parro          NUSP: 11917987
8      Lucca Tommaso Monzani                NUSP: 5342324
9
10 Arquivo header com a declaracao da classe de pedido
11 Link de estudo sobre biblioteca utility e funcao pair: https://terminalroot.com.br/2021/08/cpp-
    pair-e-tuple.html
12 */

```

```

13
14 #ifndef PEDIDO_HPP
15 #define PEDIDO_HPP
16
17 #include <string>
18 #include <vector>
19 #include <utility>
20
21 class Pedido
22 {
23     private:
24         std::string _cliente;
25         int _mesa;
26         std::vector<std::pair<std::string, int>> _itens;
27
28     public:
29         // Construtor da classe
30         Pedido(std::string cliente, int mesa);
31
32         // Metodos da classe
33         void adicionaItem(const std::string& item, int quantidade);
34         void exibirPedido() const;
35         std::vector<std::pair<std::string, int>> getPedido();
36         void finalizaPedido();
37         std::string getCliente();
38 };
39
40 #endif // PEDIDO_HPP

```

Código 12: Arquivo *pedido.hpp*.

```

1  /*
2      SCC0604 - Programacao Orientada a Objetos
3      TRABALHO PRATICO FINAL
4
5      INTEGRANTES:
6          Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7          Henrique Carobolante Parro          NUSP: 11917987
8          Lucca Tommaso Monzani               NUSP: 5342324
9
10     Arquivo source com a implementacao dos metodos publicos da classe de pedido
11     Link de estudo sobre biblioteca utility e funcao pair: https://terminalroot.com.br/2021/08/cpp-pair-e-tuple.html
12 */
13
14 #include "../header/pedido.hpp"
15
16 #include <iostream>
17
18 // Construtor da classe
19 Pedido::Pedido(std::string cliente, int mesa): _cliente(cliente), _mesa(mesa) {}
20
21 void Pedido::adicionaItem(const std::string& item, int quantidade) // Adiciona o item e a
22                             quantidade do item ao pedido
23 {
24     _itens.push_back(std::make_pair(item, quantidade));
25 }
26
27 void Pedido::exibirPedido() const // Exibe o pedido do cliente
28 {
29     std::cout << "Cliente: " << _cliente << std::endl;
30     std::cout << "Mesa: " << _mesa << std::endl;
31     std::cout << "Itens do Pedido:" << std::endl;
32     for (const auto& item : _itens)

```

```

32         std::cout << "- " << item.first << " (Quantidade: " << item.second << ")" << std::endl;
33     }
34 }
35
36 std::vector<std::pair<std::string, int>> Pedido::getPedido() // Retorna itens do pedido
37 {
38     return _itens;
39 }
40
41 void Pedido::finalizaPedido() // Finaliza o pedido
42 {
43     _itens.clear();
44 }
45
46 std::string Pedido::getClient() // Retorna o nome do cliente
47     que fez o pedido
48 {
49     return _cliente;
50 }

```

Código 13: Arquivo *pedido.cpp*.

Implementação dos arquivos *reserva.hpp* e *reserva.cpp*, respectivamente:

```

1  /*
2  SCC0604 - Programacao Orientada a Objetos
3  TRABALHO PRATICO FINAL
4
5  INTEGRANTES:
6      Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7      Henrique Carobolante Parro          NUSP: 11917987
8      Lucca Tommaso Monzani               NUSP: 5342324
9
10     Arquivo header com a declaracao da classe de reserva
11     Link de estudo sobre funcao std::tm: https://en.cppreference.com/w/cpp/chrono/c/tm
12 */
13
14 #ifndef RESERVA_HPP
15 #define RESERVA_HPP
16
17 #include <string>
18 #include <ctime>
19
20 class Reserva
21 {
22     private:
23         int _idReserva;
24         int _quantidadePessoas;
25         bool _status; // True se reserva ativa; false se reserva inativa
26         std::string _nomeCliente;
27         std::tm _dataHora;
28
29     public:
30         // Construtor da classe
31         Reserva(int idReserva, int quantidadePessoas, bool status, const std::string& nomeCliente,
32                 const std::tm& dataHora);
33
34         // Metodos da classe
35         int getIdReserva() const;
36         int getQuantidadePessoas() const;
37         bool getStatus() const;
38         std::string getNomeCliente() const;
39         std::tm getDataHora() const;
40 };

```

```
41 #endif // RESERVA_HPP
```

Código 14: Arquivo *reserva.hpp*.

```
1  /*
2  SCC0604 - Programacao Orientada a Objetos
3  TRABALHO PRATICO FINAL
4
5  INTEGRANTES:
6      Carlos Henrique Hannas de Carvalho  NUSP: 11965988
7      Henrique Carobolante Parro          NUSP: 11917987
8      Lucca Tommaso Monzani               NUSP: 5342324
9
10     Arquivo source com a implementacao dos metodos publicos da classe de reserva
11     Link de estudo sobre funcao std::tm: https://en.cppreference.com/w/cpp/chrono/c/tm
12 */
13
14 #include "../header/reserva.hpp"
15
16 // Construtor da classe
17 Reserva::Reserva(int idReserva, int quantidadePessoas, bool status, const std::string& nomeCliente,
18                 const std::tm& dataHora): _idReserva(idReserva), _quantidadePessoas(quantidadePessoas), _status(
19                 status), _nomeCliente(nomeCliente), _dataHora(dataHora) {}
20
21 int Reserva::getIdReserva() const // Retorna o ID da reserva
22 {
23     return _idReserva;
24 }
25
26 int Reserva::getQuantidadePessoas() const // Retorna a quantidade de pessoas da reserva
27 {
28     return _quantidadePessoas;
29 }
30
31 bool Reserva::getStatus() const // Retorna se a reserva esta ativa ou nao
32 {
33     return _status;
34 }
35
36 std::string Reserva::getNomeCliente() const // Retorna o nome do cliente que fez a reserva
37 {
38     return _nomeCliente;
39 }
40
41 std::tm Reserva::getDataHora() const // Retorna a data e horario da reserva
42 {
43     return _dataHora;
44 }
```

Código 15: Arquivo *reserva.cpp*.