Programação Orientada aos Objetos

Financial Services

Docente:
Fernando José Barros Rodrigues da Silva

Membros:
Cintia Cumbane(2020244607)
Cristiana Gonçalves(2019239753)

Dezembro,2024

ÍNDICE

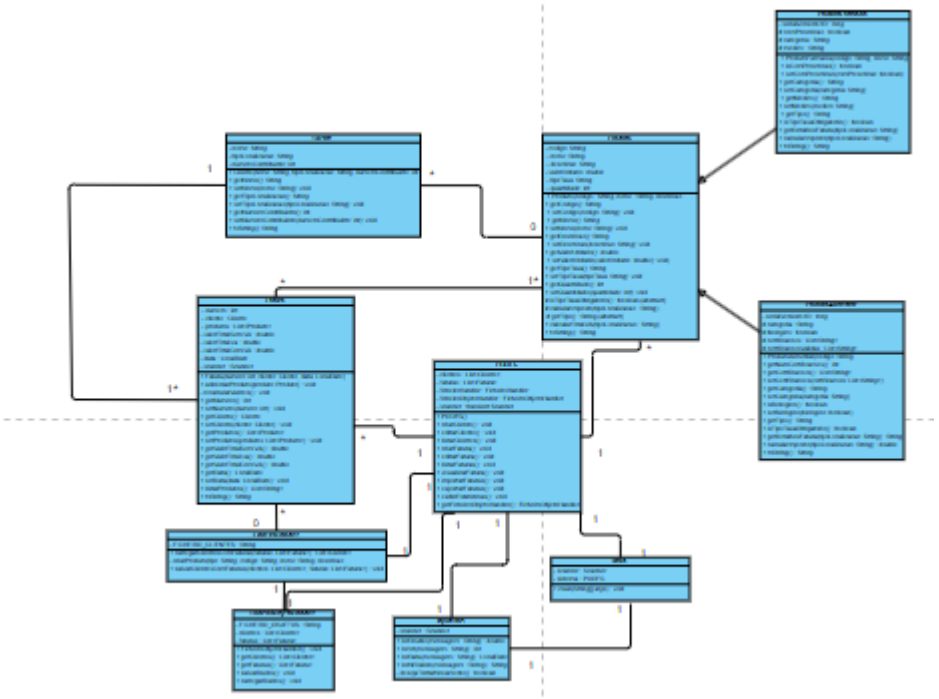
Introdução

O projeto implementado é uma aplicação de gestão financeira denominada **POO Financial Services (POOFS)**, desenvolvida para facilitar o registo e o controlo de faturas emitidas por uma empresa, com funcionalidades que permitem a exportação para o portal das Finanças. Esta aplicação é projetada para lidar com diferentes tipos de produtos, como alimentares e farmacêuticos, e gerenciar as diversas taxas de IVA e condições fiscais associadas, que variam de acordo com a localização do cliente e as características específicas de cada produto.

A aplicação foi desenvolvida utilizando a linguagem Java, seguindo uma abordagem de **Programação Orientada a Objetos (POO)**, com ênfase em conceitos fundamentais como **herança**, **polimorfismo** e **encapsulamento**. Isso permitiu a criação de uma estrutura modular e flexível, onde diferentes tipos de produtos (alimentares e farmacêuticos) podem ser tratados de forma especializada, mas ainda mantendo a uniformidade na gestão de dados e operações.

Além disso, o projeto incorpora a **persistência de dados**, garantindo que as informações sobre clientes, faturas e produtos sejam armazenadas permanentemente. Para isso, utilizamos dois métodos principais de armazenamento: **ficheiros de texto** (para uma leitura e escrita legível) e **ficheiro de objeto** (para salvar dados de forma eficiente em formato binário). Esse processo de persistência assegura que os dados não sejam perdidos entre as execuções do programa, permitindo que sejam recuperados sempre que o sistema for reiniciado.

Diagrama UML



Desenvolvimento das Classes

1. Cliente

A classe Cliente é utilizada para representar um cliente de forma segura e validada em sistemas que requerem manipulação de dados pessoais ou fiscais.
Seguem os aspectos principais desta implementação:

Atributos

- **nome**: Armazena o nome do cliente.
- **tipoLocalizacao**: Indica a localização do cliente, que deve ser um dos seguintes valores: "Continente", "Madeira" ou "Açores".
- **numeroContribuinte**: Representa o número fiscal do cliente.

Métodos

- **Construtor**: Inicializa os atributos nome, tipoLocalizacao e numeroContribuinte. As validações são realizadas nos métodos setters dos respectivos atributos.
- **getNome() / setNome(String)**: Obtém ou define o nome do cliente.
- **getTipoLocalizacao() / setTipoLocalizacao(String)**:
  - ➔ Valida se o valor fornecido corresponde a "Continente", "Madeira" ou "Açores"
  - ➔ Remove espaços em branco desnecessários do valor fornecido (usando `trim()`).
  - ➔ Lança uma exceção (IllegalArgumentException) caso o valor seja inválido.
- **getNumeroContribuinte() / setNumeroContribuinte(int)**:
  - ➔ Garante que o número de contribuinte seja superior a zero.
  - ➔ Lança uma exceção (IllegalArgumentException) caso o valor seja inválido.
- **toString()**: Retorna uma representação textual dos atributos do cliente. Este método é útil para a exibição de informações.

Validações

- **tipoLocalizacao**: Deve ser um dos valores permitidos: "Continente", "Madeira" ou "Açores". Valores nulos ou diferentes dos especificados resultam numa exceção.
- **numeroContribuinte**: Deve ser um número inteiro positivo. Valores iguais ou inferiores a zero geram uma exceção.

Notas Adicionais

- A classe implementa a interface **Serializable**, permitindo que objetos da classe sejam convertidos para uma sequência de bytes e armazenados.
- A constante **serialVersionUID** foi definida para garantir compatibilidade durante a serialização.

2. Produto

A classe Produto é uma classe abstrata, define a estrutura e o comportamento base de um produto no sistema. É projetada para ser estendida por subclasses que implementam os métodos abstratos de acordo com os requisitos específicos de cada tipo de produto.
Seguem os aspectos principais desta implementação:

Atributos

- **codigo**: Identificador único do produto.
- **nome**: Nome do produto.
- **descricao**: Descrição detalhada do produto.
- **valorUnitario**: Valor unitário do produto sem IVA, que deve ser positivo.
- **tipoTaxa**: Tipo de taxa aplicável ao produto. Pode ser "Taxa reduzida", "Taxa intermédia" ou "Taxa normal".
- **quantidade**: Quantidade disponível do produto. Deve ser um número não negativo.

Métodos

➔ **Construtor**: Inicializa os atributos principais, com validação dos valores para valorUnitario, tipoTaxa e quantidade.
- **getCodigo() / setCodigo(String)**: Obtém ou define o código do produto.
- **getNome() / setNome(String)**: Obtém ou define o nome do produto.
- **getDescricao() / setDescricao(String)**: Obtém ou define a descrição do produto.
- **getValorUnitario() / setValorUnitario(double)**:
  - ➔ Valida que o valor unitário seja positivo.
  - ➔ Lança exceção (IllegalArgumentException) caso o valor seja inválido.
- **getTipoTaxa() / setTipoTaxa(String)**:
  - ➔ Valida se o tipo de taxa é "Taxa reduzida", "Taxa intermédia" ou "Taxa normal".
  - ➔ Lança exceção caso o valor seja inválido.
- **getQuantidade() / setQuantidade(int)**:
  - ➔ Assegura que a quantidade seja não negativa.
  - ➔ Lança exceção caso o valor seja inválido.
- **calcularTotalIVA(String tipoLocalizacao)**: Calcula o valor total de IVA aplicável ao produto com base na quantidade e no tipo de localização do cliente.
- **toString()**: Retorna uma representação textual dos atributos do produto para exibição ou depuração.

Métodos abstratos:

- **isTipoTaxaObrigatorio()**: Determina se o atributo tipoTaxa é obrigatório para o produto.
- **calcularImposto(String tipoLocalizacao)**: Calcula o imposto aplicável ao produto com base no tipo de localização do cliente.
- **getCategoria()**: Retorna a categoria do produto.
- **getDetalhesFatura(String tipoLocalizacao)**: Retorna os detalhes específicos do produto para emissão de faturas.

- **getTipo()**: Retorna o tipo específico do produto.

Validações

- **valorUnitario**: Deve ser maior que zero. Valores inválidos resultam numa exceção.
- **tipoTaxa**: Deve ser uma das opções válidas ("Taxa reduzida", "Taxa intermédia" ou "Taxa normal"). Caso contrário, lança exceção.
- **quantidade**: Não pode ser negativa. Valores inválidos lançam uma exceção.

Notas Adicionais

- A classe implementa **Serializable** com um **serialVersionUID**, permitindo salvar e carregar objetos do tipo **Produto**.

3. ProdutoAlimentar

A classe ProdutoAlimentar é uma subclasse especializada de Produto, com foco em produtos alimentares. Ela valida dados como o tipo de taxa, certificação e categoria (ex.: "congelados", "vinho"), além de permitir ajustes nos cálculos de impostos com base em características como ser biológico, ter certificações e qual a sua categoria.

Atributos:

- **categoria**: Define a categoria do produto alimentar, como "congelados", "enlatados" ou "vinho".
- **biologico**: Indica se o produto é biológico. Onde esta informação afetará o cálculo de impostos
- **certificacoes**: Lista das certificações do produto (como "ISO22000", "HACCP", etc.).
- **certificacoesValidas**: Lista pré-definida de certificações válidas.

Construtor:

➔ O construtor inicializa todos os atributos necessários, validando as certificações e a categoria do produto com base no tipo de taxa especificado.

Métodos Específicos:

- **getNumCertificacoes()**: Retorna o número de certificações do produto.
- **getCertificacoes()**: Retorna a lista de certificações do produto.
- **setCertificacoes()**: Válida e define as certificações, dependendo do tipo de taxa do produto. Para "Taxa Reduzida", as certificações são obrigatórias e limitadas entre 1 e 4. Para "Taxa Intermédia" e "Taxa Normal", o produto não pode ter certificações.
- **getCategoria()** e **setCategoria()**: Define e valida a categoria do produto, dependendo do tipo de taxa. Produtos com "Taxa Reduzida" não podem ter categoria específica, enquanto produtos com "Taxa Intermédia" devem pertencer a categorias específicas.
- **isBiologico()** e **setBiologico()**: Define se o produto é biológico.
- **getTipo()**: Retorna o tipo do produto, que é "Alimentar" para esta classe.

- **isTipoTaxaObrigatorio()**: Retorna true, indicando que a taxa é obrigatória para produtos alimentares.

Métodos Abstratos Implementados:

- **getDetalhesFatura(String tipoLocalizacao)**: Retorna informações específicas do produto, como se é biológico e o número de certificações, para exibição na fatura.
- **calcularImposto(String tipoLocalizacao)**: Calcula o valor do imposto para o produto com base no tipo de localização e tipo de taxa (e.g., "Taxa Reduzida", "Taxa Intermédia" ou "Taxa Normal"). Considera descontos ou aumentos dependendo de características como "biológico", número de certificações e categorias.

Regra de Cálculo de Impostos:

O cálculo do imposto é baseado na localização do cliente e no tipo de taxa. Dependendo da localização (Continente, Madeira ou Açores), a taxa de IVA varia. O imposto também pode ser ajustado com base em características do produto, como:

- **Biológico**: Desconto de 10% na taxa de IVA.
- **Certificações**: Redução de 1% na taxa de IVA se o produto tiver 4 certificações.
- **Categoria "Vinho"**: Aumento de 1% na taxa de IVA para produtos dessa categoria.

Método toString():

Retorna uma representação em formato de string de todos os atributos do produto, incluindo código, nome, descrição, valor unitário, tipo de taxa, se é biológico, certificações, categoria e quantidade disponível.

4. ProdutoFarmacia

A classe ProdutoFarmacia é uma subclasse de **Produto**, focada em representar produtos farmacêuticos com regras e características específicas, como prescrição médica, categorias de produtos e cálculos de impostos.

Atributos

- **comPrescricao**: Indica se o produto necessita de prescrição médica.
- **categoria**: Categoria do produto, válida apenas para produtos sem prescrição, podendo ser "Beleza", "Bem-estar", "Bebês", "Animais", ou "Outro".
- **medico**: Nome do médico responsável pela prescrição, aplicável somente a produtos com prescrição médica.

Construtor:

➔ Inicializa todos os atributos necessários, realizando validações específicas:
- Se o produto for com prescrição, valida o nome do médico e desabilita a categoria.

- Se o produto não for com prescrição, valida a categoria conforme uma lista predefinida e desabilita o campo do médico.

Métodos Específicos:

- **getTipo()**: Retorna "Farmacia", indicando que o tipo do produto é farmacêutico.
- **isTipoTaxaObrigatorio()**: Retorna false, pois a taxa não é obrigatória para produtos de farmácia.
- **getDetalhesFatura(String tipoLocalizacao)**: Retorna detalhes para a fatura, mostrando se o produto é com prescrição e, em caso afirmativo, o nome do médico ou, caso contrário, a categoria do produto.
- **calcularImposto(String tipoLocalizacao)**: Calcula o valor do imposto com base na localização (Continente, Madeira, ou Açores) e no tipo de produto. Para produtos com prescrição, a taxa de IVA é reduzida. Para produtos sem prescrição, aplica-se uma redução de 1% na categoria "Animais". A taxa nunca será negativa.
- **toString()**: Retorna uma representação textual do produto com todos os atributos, incluindo o código, nome, descrição, valor unitário, tipo de taxa, prescrição, categoria, médico (se aplicável) e quantidade disponível.

Validação de Categoria

➔ A categoria é validada com base em um conjunto de valores pré-definidos para produtos sem prescrição, garantindo que a categoria seja uma das opções permitidas: "Beleza", "Bem-estar", "Bebês", "Animais", ou "Outro".

Regra de Cálculo de Impostos:

➔ O cálculo do imposto depende da localização do cliente:

- Continente: 6% para produtos com prescrição, 23% para produtos sem prescrição.
- Madeira: 5% para produtos com prescrição, 23% para produtos sem prescrição.
- Açores: 4% para produtos com prescrição, 23% para produtos sem prescrição.
- Se a categoria for "Animais" e o produto não for com prescrição, a taxa é reduzida em 1%.
- O imposto é sempre garantido para ser positivo, usando a função Math.max() para evitar valores negativos.

Método toString():

➔ Retorna uma string formatada que inclui os principais atributos do produto, como código, nome, descrição, valor unitário, tipo de taxa, prescrição, categoria, médico e quantidade disponível.

5. Fatura

A classe **Fatura** representa a estrutura de dados para registrar informações relacionadas a uma compra realizada por um cliente. Ela associa produtos adquiridos ao cliente e realiza cálculos de valores totais da compra, incluindo o imposto sobre valor agregado (IVA).

Além disso, fornece métodos para manipular e acessar os dados, garantindo integridade e facilidade de uso.

Atributos:

- **numero**: Número identificador único da fatura.
- **cliente**: Cliente associado à fatura.
- **produtos**: Lista de produtos adquiridos na fatura.
- **valorTotalSemIVA**: Valor total dos produtos sem considerar o IVA.
- **valorTotalIva**: Valor total do IVA dos produtos.
- **valorTotalComIVA**: Valor total dos produtos incluindo o IVA.
- **data**: Data de emissão da fatura.
- **scanner** (transiente): Objeto utilizado para entrada de dados interativos (não persistido durante serialização).

Construtor:

- ➔ O construtor inicializa a fatura com um número identificador, o cliente associado e a data de emissão.
- numero: Identificação numérica da fatura.
- cliente: O cliente que realiza a compra.
- data: Data de emissão da fatura.
- A lista de produtos é inicializada como uma nova lista vazia.

Métodos Específicos:

- **adicionarProduto(Produto produto)**: Adiciona um produto à fatura e recalcula os valores totais (sem IVA, IVA e com IVA).
- **recalcularValores()**: Atualiza os valores totais da fatura (sem IVA, IVA e com IVA) com base nos produtos adicionados. Este método é chamado automaticamente sempre que a lista de produtos ou o cliente é alterado.
- **listarProdutos()**: Retorna uma lista de Strings, representando os produtos incluídos na fatura. Cada produto é representado como uma string através do método toString() da classe Produto.

. Getters e Setters:

- **getNumero()** e **setNumero(int numero)**: Retornam e definem o número da fatura.
- **getCliente()** e **setCliente(Cliente cliente)**: Retornam e definem o cliente associado à fatura. Caso o cliente seja alterado, os valores da fatura são recalculados.
- **getProdutos()**: Retorna uma cópia imutável da lista de produtos, garantindo que a lista interna não seja modificada externamente.
- **setProdutos(List<Produto> produtos)**: Substitui a lista de produtos e recalcula os valores totais da fatura.
- **getValorTotalSemIVA(), getValorTotalIva(), getValorTotalComIVA()**: Retornam os valores totais da fatura, respectivamente, sem IVA, apenas o IVA, e com IVA.

- **getData()** e **setData(LocalDate data)**: Retornam e definem a data de emissão da fatura.

Métodos Abstratos Implementados:

- **toString()**: Retorna uma representação textual da fatura, incluindo todos os atributos da classe, como número da fatura, cliente, produtos, valores totais (sem IVA, IVA e com IVA) e a data de emissão.

Lógica de Cálculo de Valores:

➔ A fatura recalcula os valores totais sempre que um produto é adicionado ou o cliente é alterado.
- **Valor sem IVA**: É calculado multiplicando o valor unitário de cada produto pela sua quantidade.
- **Valor do IVA**: Calculado chamando o método calcularImposto() de cada produto, passando a localização do cliente como parâmetro.
- **Valor com IVA**: É a soma do valor sem IVA e o valor do IVA.

Métodos Adicionais:

- **listarProdutos()**: Este método retorna uma lista de representações de string dos produtos na fatura, usando o método toString() de cada produto.

Notas Adicionais:

- A classe implementa **Serializable**, permitindo que os objetos de **Fatura** sejam persistidos.
- O atributo **scanner** é marcado como **transient**, sendo excluído durante a serialização, já que não é necessário para persistência.

6. POOFS

A classe gerencia faturas de clientes, permitindo criar, editar, listar, visualizar, importar e exportar faturas. Ela organiza **clientes** e **faturas** em listas e trata diferentes tipos de **produtos** (alimentares e farmacêuticos). A classe também lida com a persistência de dados, salvando e carregando informações de arquivos

A classe possui os seguintes atributos privados:

- **clientes** (List<Cliente>): Lista de objetos do tipo Cliente, que armazenam as informações dos clientes.
- **faturas** (List<Fatura>): Lista de objetos do tipo Fatura, onde cada fatura representa uma transação ou compra realizada por um cliente.
- **ficheiroHandler** (FicheiroHandler): Responsável por realizar operações de leitura e escrita de dados em arquivos. Presume-se que esse handler manipule arquivos de dados para armazenar ou carregar clientes e faturas.

- **ficheiroObjetoHandler** (FicheiroObjetoHandler):Tem como objetivo salvar ou carregar listas de objetos (como Cliente e Fatura) em arquivos binários ou texto.
- **scanner** (Scanner): Um objeto Scanner para leitura de dados do usuário via terminal. Ele é utilizado para capturar entradas do usuário para processar informações como faturas, produtos, etc.

Métodos

1. **criarFatura()**

- Este método permite a criação de uma nova fatura para um cliente.
- Primeiro, o método solicita informações do cliente, como tipo de local e número de contribuinte.
- A seguir, o usuário insere os produtos que irão compor a fatura (cada produto pode ser de tipo "Alimentar" ou "Farmácia"), e os detalhes relacionados a cada tipo de produto são coletados, como categoria, taxa de IVA, certificações, entre outros.
- No final, a fatura é adicionada à lista de faturas, e a lista de clientes e faturas é salva por meio do ficheiroHandler.

2. **editarFatura()**:

- O método permite a edição de uma fatura existente.
- O usuário pode alterar a data da fatura e adicionar novos produtos.
- O código segue uma lógica semelhante ao método criarFatura(), mas, no caso da edição, ele primeiro localiza a fatura existente pela chave (número da fatura).
- Também é possível alterar as informações dos produtos da fatura.
- No final, as alterações são salvas novamente no arquivo por meio do ficheiroHandler.

3. **listarFaturas()**:

- Exibe um resumo de todas as faturas registradas.
- Para cada fatura, são apresentados detalhes como número da fatura, cliente, local de faturamento, quantidade de produtos e valores totais com e sem IVA.

4. **visualizarFatura()**:

- Esse método permite visualizar os detalhes completos de uma fatura específica, incluindo dados do cliente, produtos e valores de IVA.
- Além disso, a tabela de produtos exibe detalhes como o código, nome, descrição, categoria, quantidade, valor unitário, IVA, e outras informações.

5. **importarFaturas()**:

- Este método importa faturas e clientes de um arquivo externo, atualizando as listas de faturas e clientes no sistema.
- Ele usa o ficheiroHandler para carregar os dados e integra esses dados ao sistema atual.

6. **exportarFaturas()**:

- Permite exportar as faturas para um arquivo de texto, onde cada fatura é registrada com os seus detalhes, incluindo os produtos associados.
- O arquivo gerado pode ser salvo em um diretório específico indicado pelo usuário.

7. **exibirEstatisticas()**:
- Exibeestatísticas sobre as faturas registradas, como o número total de faturas, o número de produtos vendidos, os valores totais sem e com IVA.
- Isso fornece uma visão geral do desempenho do sistema em termos de vendas.
8. **getFicheiroObjetoHandler()**:
- Retorna o objeto ficheiroObjetoHandler, permitindo que outras partes do programa possam acessar e manipular o comportamento do ficheiro de objetos.

Lógica Geral

1. **Produtos Alimentares**:
- Para os produtos alimentares, o programa permite definir diferentes tipos de taxa (reduzida, intermediária ou normal).
- Dependendo do tipo de taxa, o produto pode ter diferentes características, como ser "biológico" ou ter certificações (ISO22000, HACCP, etc.).
- A categoria também pode ser configurada (ex: congelados, enlatados, etc.).

2. **Produtos de Farmácia**:
- Para produtos de farmácia, o usuário pode definir se o produto exige prescrição médica ou se pertence a uma categoria como beleza, bem-estar, bebês, etc.
- Se houver prescrição, o nome do médico que prescreveu o produto deve ser informado.

7. FicheiroHandler

A classe FicheiroHandler é responsável pela persistência de dados em um sistema orientado a objetos, permitindo que informações sobre **clientes**, **faturas** e **produtos** sejam salvas em um arquivo (clientes.txt) e recuperadas entre execuções do programa.

Atributos:

- **FICHEIRO_CLIENTES**: Caminho para o arquivo de texto onde os dados serão armazenados ou carregados. Nesse caso, o arquivo é chamado **clientes.txt**.

Métodos Principais:

➔ **carregarClientesComFaturas(List<Fatura> faturas)**

Esse método é responsável por carregar os dados de clientes e faturas a partir do arquivo **clientes.txt** e preencher ou adicionar às listas de **clientes** e **faturas**.

- **Carregar dados de clientes**: Lê os dados de clientes (nome, tipo de localização, número de contribuinte) e cria objetos Cliente, adicionando-os à lista de clientes.

- **Carregar faturas e produtos**: Quando a leitura do arquivo atinge a seção de faturas, o método lê as faturas (número, data e número de contribuinte) e cria objetos Fatura, associando-os aos clientes correspondentes.
- **Criar produtos**: Para cada fatura, lê os produtos (código, nome, descrição, tipo, valor unitário, quantidade) e cria objetos Produto, ProdutoAlimentar ou ProdutoFarmacia, dependendo do tipo de produto.

➔ **criarProduto(...)**

Este método cria um produto com base nas informações do arquivo, levando em consideração o tipo do produto (alimentar ou farmacêutico). Ele válida e normaliza os dados antes de criar o objeto correto:

- **Produto alimentar**: Verifica se o produto é biológico, se tem certificações e ajusta a taxa de imposto (reduzida, normal ou intermédia).
- **Produto de farmácia**: Verifica se o produto requer prescrição e válida a categoria, dependendo se o produto tem ou não prescrição.

➔ **salvarClientesComFaturas(List<Cliente> clientes, List<Fatura> faturas)**

Esse método salva os dados de clientes e faturas em um arquivo **clientes.txt**.

- **Clientes**: Salva as informações dos clientes no formato nome; tipoLocalizacao; numeroContribuinte.
- **Faturas e produtos**: Salva as faturas no formato numeroFatura; data; numeroContribuinte e, em seguida, para cada fatura, salva os produtos associados no formato codigo; nome; descricao; tipo; valorUnitario; quantidade; categoria.
    - Para **produtos alimentares**, salva se o produto é biológico e suas certificações.
    - Para **produtos de farmácia**, salva se o produto requer prescrição e o nome do médico, se presente.

Mecanismos de Persistência:

- **Carregar dados**: Quando o sistema é iniciado, a classe lê o arquivo **clientes.txt** para carregar os dados de clientes e faturas que foram armazenados na execução anterior.
- **Salvar dados**: Ao final da execução do programa ou em momentos específicos, os dados de clientes, faturas e produtos são salvos de volta no arquivo, garantindo que as informações possam ser persistidas entre diferentes execuções do programa.

Estrutura do Arquivo:

O arquivo **clientes.txt** segue uma estrutura de seções:

- **# Clientes**: Contém as informações dos clientes.

- **# Faturas**: Contém as faturas, seguidas dos produtos associados a cada fatura.
- As linhas são separadas por ponto e vírgula (;), e as informações são lidas e processadas para criar as instâncias correspondentes de Cliente, Fatura, Produto, ProdutoAlimentar, ou ProdutoFarmacia.

Tratamento de Erros:

A classe trata de possíveis erros durante a leitura e escrita de arquivos:

- **IOException**: Caso ocorra algum erro ao acessar o arquivo.
- **Exceções gerais**: Captura erros inesperados e fornece mensagens apropriadas.

Benefícios do Uso da Persistência de Dados:

- **Armazenamento permanente**: Garante que as informações dos clientes e faturas sejam mantidas entre as execuções do programa.
- **Escalabilidade**: A abordagem com listas permite que o sistema cresça e se adapte a um número maior de clientes, faturas e produtos.
- **Facilidade de manutenção**: A separação da lógica de leitura e escrita de dados torna o código modular e fácil de manter.

8. FicheiroObjetosHandler

FicheiroObjetoHandler oferece uma solução robusta para persistência de dados, permitindo salvar e carregar informações complexas, como listas de clientes e faturas, em um arquivo binário. Essa abordagem garante integridade, eficiência e facilidade na manipulação de dados persistentes.

Atributos:

- **clientes**: Lista de objetos Cliente, representando os clientes armazenados.
- **faturas**: Lista de objetos Fatura, representando as faturas associadas aos clientes.
- **FICHEIRO_OBJETOS**: Constante que define o nome do arquivo binário usado para persistência dos dados ("dados.obj").

Métodos:

- **FicheiroObjetoHandler()**: Construtor que inicializa as listas de clientes e faturas, e chama o método carregarDados() para carregar os dados persistidos ao inicializar o handler.
- **getClientes()**: Retorna a lista de clientes carregados.
- **getFaturas()**: Retorna a lista de faturas carregadas.
- **salvarDados()**: Salva as listas de clientes e faturas no arquivo binário dados.obj. Utiliza a serialização para gravar os objetos.
- **carregarDados()**: Carrega as listas de clientes e faturas do arquivo binário. Caso o arquivo não exista, ele cria um novo e deixa as listas vazias

Funcionamento:

- **Persistência com serialização**: A classe utiliza o ficheiro de objetos(ObjectOutputStream e ObjectInputStream) para salvar e carregar os dados, permitindo armazenar objetos complexos, como listas de clientes e faturas, de forma eficiente.
- **Criação de arquivo vazio**: Se o arquivo de objetos não for encontrado, ele cria um novo arquivo vazio e salva as listas.
- **Leitura e escrita de objetos**: Utiliza fluxos de entrada e saída de objetos (ObjectInputStream e ObjectOutputStream) para manipular os dados persistentes.

9. Main

O main tem como objetivo no nosso projeto a manipulação de clientes, faturas e suas respectivas operações (como criar, editar, listar, visualizar, importar, exportar e exibir estatísticas). A estrutura central é um **menu interativo**, no qual o usuário escolhe a operação a ser realizada.

Resumo das funcionalidades do código:

1. **Menu Interativo**: O programa exibe um menu com 11 opções para o usuário escolher. As opções incluem:

➔ Criar e editar clientes
➔ Listar clientes
➔ Criar, editar e listar faturas
➔ Visualizar faturas
➔ Importar e exportar faturas
➔ Exibir estatísticas
➔ Sair do programa

2. **Validação de Entrada**:

➔ O código captura as opções digitadas pelo usuário é válida se o número inserido está dentro do intervalo correto (1 a 11).
➔ Caso o usuário insira um valor inválido (não numérico ou fora do intervalo), o sistema exibe uma mensagem de erro e solicita que o usuário tente novamente.

3. **Estrutura de Controle**:

➔ Um switch é usado para decidir qual ação o sistema executa com base na opção escolhida pelo usuário.
➔ Cada caso no switch chama um método correspondente do objeto sistema (instância da classe POOFS).
➔ Se a opção for **11 (Sair)**, o programa salva os dados usando o salvarDados() e encerra a execução.

Detalhamento das Ações no Menu

➔ Cada opção vai chamar um método correspondente para tratar da respectiva funcionalidade:

1. **Criar Cliente**: método criarCliente() no objeto sistema para permitir que o usuário registre um novo cliente.
2. **Editar Cliente**: método editarCliente() no objeto sistema para editar as informações de um cliente existente.
3. **Listar Clientes**: método listarClientes() para exibir todos os clientes registrados.
4. **Criar Fatura**: método criarFatura() para criar uma nova fatura.
5. **Editar Fatura**: método editarFatura() para editar uma fatura existente.
6. **Listar Faturas**: método listarFaturas() para exibir todas as faturas registradas.
7. **Visualizar Fatura**: método visualizarFatura**()** para exibir detalhes de uma fatura específica.
8. **Importar Faturas**: método importarFaturas() para importar faturas de um arquivo.
9. **Exportar Faturas**: método exportarFaturas() para exportar as faturas para um arquivo.
10. **Exibir Estatísticas**: método exibirEstatisticas() para mostrar informações analíticas ou estatísticas sobre o sistema.
11. **Sair**: Salva todos os dados (clientes e faturas) usando o método salvarDados() do FicheiroObjetoHandler e encerra o programa.

10. InputUtils

A classe InputUtils fornece métodos auxiliares para a leitura e validação de entradas do usuário, garantindo que as entradas sejam válidas e tratando erros de forma adequada. Ela utiliza um único objeto `Scanner` para capturar entradas do console, o que facilita a reutilização e o controle das entradas durante a execução do programa.

Funcionalidades:

1. **Leitura de números decimais (`double`)**
   O método `lerDouble(String mensagem)` solicita ao usuário que insira um número do tipo `double`. Caso a entrada não seja válida (não numérica), o método exibe uma mensagem de erro e pergunta ao usuário se deseja tentar novamente. Se o usuário decidir cancelar, a operação é interrompida com uma exceção.
2. **Leitura de números inteiros (`int`)**
   O método `lerInt(String mensagem)` funciona de maneira similar ao `lerDouble()`, mas para a entrada de números inteiros. Ele valida a entrada, exibe um erro em caso de falha e permite ao usuário tentar novamente ou cancelar a operação.
3. **Leitura de datas (`LocalDate`)**
   O método `lerData(String mensagem)` solicita ao usuário que insira uma data no formato `yyyy-MM-dd`. Ele tenta converter a string de entrada em um objeto

`LocalDate`. Se o formato da data for inválido, uma mensagem de erro é exibida, e o usuário pode escolher tentar novamente ou cancelar a operação.

4. **Leitura de NIF (Número de Identificação Fiscal) válido**
   O método `lerNifValido(String mensagem)` solicita ao usuário que insira um NIF com exatamente 4 dígitos. Caso a entrada não corresponda ao formato esperado (4 dígitos numéricos), o método exibe um erro e permite ao usuário tentar novamente ou cancelar a operação.
5. **Método auxiliar para tentar novamente**
   O método `desejaTentarNovamente()` é usado para perguntar ao usuário se ele deseja tentar novamente após uma entrada inválida. O usuário pode responder com "s" para sim ou "n" para não. Se a resposta for "n", a operação é cancelada e uma exceção é lançada.

11. Cliente.txt

O ficheiro clientes.txt serve para **persistir dados** no sistema, ou seja, para armazenar permanentemente as informações sobre os clientes e as faturas de forma que o sistema possa recarregar esses dados entre as execuções do programa. A persistência garante que os dados não sejam perdidos após o fechamento do sistema e possam ser utilizados novamente na próxima vez que o programa for executado.

Clientes: A primeira parte do ficheiro contém dados sobre os clientes. Esses dados são essenciais para identificar quem está adquirindo produtos ou serviços, e o número de contribuinte é um identificador único para associar faturas a um cliente específico.
Faturas: A segunda parte do ficheiro contém os registos de faturas. Cada fatura é associada a um cliente e contém informações sobre os produtos adquiridos. Isso inclui detalhes sobre o tipo de produto (alimentar ou farmácia), o preço, a quantidade e outras características, como se o produto é biológico ou se exige prescrição médica.

11. Fatura.txt

Esse ficheiro garante que as transações realizadas pelo cliente sejam armazenadas permanentemente, mesmo após o sistema ser fechado. Quando o sistema é reiniciado, os dados das faturas podem ser carregados novamente para que o histórico de compras e a contabilidade sejam mantidos.
A primeira linha de cada seção de fatura contém o **número da fatura**, a **data** de emissão e o **número de contribuinte** do cliente.
As linhas subsequentes contêm os produtos comprados, com informações detalhadas, como o **código do produto**, o **nome**, a **descrição**, o **tipo** de produto, o **valor unitário**, a **quantidade** adquirida e, se aplicável, informações sobre a **categoria**, **biológico** e **certificações** no caso de produtos alimentares, ou **prescrição médica** e **médico** para produtos farmacêuticos.

Conclusão

O projeto POOFS é uma aplicação eficaz de princípios essenciais da Programação Orientada a Objetos (POO), como encapsulamento, modularidade e persistência de dados.

A utilização de classes como Cliente, Fatura e Produto facilita a organização dos dados e a realização de operações de forma clara e eficiente. A persistência foi tratada por meio da combinação de ficheiros de texto e de objeto, permitindo que os dados sejam armazenados e recuperados de maneira eficaz entre sessões, o que assegura a integridade das informações ao longo do tempo.