Explicação Detalhada do Código ERP Refatorado

Este documento tem como objetivo explicar em detalhes o código do sistema ERP refatorado, abordando a funcionalidade de cada arquivo, classe, método e, quando relevante, linha por linha.

1. `database/database_manager.py`

Este arquivo é responsável por gerenciar a conexão com o banco de dados SQLite e garantir que todas as tabelas necessárias para o sistema ERP sejam criadas. Ele utiliza o padrão de *context manager* para um gerenciamento seguro e eficiente da conexão com o banco de dados.

```python
import sqlite3

class DatabaseManager:
    def __init__(self, db_name="clientes.bd"):
        self.db_name = db_name

    def __enter__(self):
        self.conn = sqlite3.connect(self.db_name)
        self.cursor = self.conn.cursor()
        return self.cursor

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.conn.commit()
        self.conn.close()

    def create_tables(self):
        with self as cursor:
            cursor.execute("""
                CREATE TABLE IF NOT EXISTS usuarios(
                    id_usuario INTEGER PRIMARY KEY AUTOINCREMENT,
                    nome_usuario TEXT NOT NULL,
                    cpf_usuario TEXT NOT NULL UNIQUE,
                    email_usuario TEXT NOT NULL UNIQUE,
                    telefone_usuario TEXT NOT NULL,
                    data_nascimento TEXT NOT NULL,
                    rua TEXT NOT NULL,
                    cep TEXT NOT NULL,
                    bairro TEXT NOT NULL,
                    cidade TEXT NOT NULL,
                    senha TEXT NOT NULL,
                    tipo TEXT NOT NULL CHECK(tipo IN (
                        'admin', 'vendedor', 'financeiro', 'estoque'
                    )),
                    permissao TEXT NOT NULL DEFAULT 'padrao'
                )
            """)

            cursor.execute("""
                CREATE TABLE IF NOT EXISTS clientes (
                    id_cliente INTEGER PRIMARY KEY AUTOINCREMENT,
                    nome_cliente TEXT NOT NULL,
                    cpf_cliente TEXT NOT NULL UNIQUE,
                    email_cliente TEXT NOT NULL UNIQUE,
                    telefone_cliente TEXT NOT NULL,
                    data_nascimento TEXT NOT NULL,
                    rua TEXT NOT NULL,
                    cep TEXT NOT NULL,
                    bairro TEXT NOT NULL,
                    cidade TEXT NOT NULL
                )
            """)

            cursor.execute("""
                CREATE TABLE IF NOT EXISTS fornecedores (
                    id_fornecedor INTEGER PRIMARY KEY AUTOINCREMENT,
                    nome TEXT NOT NULL,
                    cnpj TEXT NOT NULL UNIQUE,
                    telefone TEXT,
                    email TEXT UNIQUE,
                    rua TEXT NOT NULL,
                    cep TEXT NOT NULL,
```

```
                    bairro TEXT NOT NULL,
                    cidade TEXT NOT NULL
                )
            """)

            cursor.execute("""
                CREATE TABLE IF NOT EXISTS produtos (
                    id_produto INTEGER PRIMARY KEY AUTOINCREMENT,
                    nome TEXT NOT NULL,
                    descricao TEXT,
                    preco_venda REAL NOT NULL,
                    preco_compra REAL NOT NULL,
                    fornecedor_id INTEGER,
                    FOREIGN KEY(fornecedor_id) REFERENCES
fornecedores(id_fornecedor)
                )
            """)

            cursor.execute("""
                CREATE TABLE IF NOT EXISTS estoque (
                    id_estoque INTEGER PRIMARY KEY AUTOINCREMENT,
                    produto_id INTEGER NOT NULL,
                    quantidade INTEGER NOT NULL DEFAULT 0,
                    FOREIGN KEY(produto_id) REFERENCES produtos(id_produto)
                )
            """)

            cursor.execute("""
                CREATE TABLE IF NOT EXISTS vendas (
                    id_vendas INTEGER PRIMARY KEY AUTOINCREMENT,
                    cliente_id INTEGER NOT NULL,
                    usuario_id INTEGER NOT NULL,
                    data_venda TEXT NOT NULL DEFAULT CURRENT_TIMESTAMP,
                    total REAL NOT NULL,
                    FOREIGN KEY(cliente_id) REFERENCES clientes(id_cliente),
                    FOREIGN KEY(usuario_id) REFERENCES usuarios(id_usuario)
                )
            """)

            cursor.execute("""
                CREATE TABLE IF NOT EXISTS compras (
                    id_compras INTEGER PRIMARY KEY AUTOINCREMENT,
                    fornecedor_id INTEGER NOT NULL,
                    usuario_id INTEGER NOT NULL,
                    data_compra TEXT NOT NULL DEFAULT CURRENT_TIMESTAMP,
                    total REAL NOT NULL,
                    FOREIGN KEY(fornecedor_id) REFERENCES
fornecedores(id_fornecedor),
                    FOREIGN KEY(usuario_id) REFERENCES usuarios(id_usuario)
                )
            """)

            cursor.execute("""
                CREATE TABLE IF NOT EXISTS financeiro (
                    id_financeiro INTEGER PRIMARY KEY AUTOINCREMENT,
                    tipo TEXT NOT NULL CHECK(tipo IN (
                        'entrada', 'saida'
                    )),
                    valor REAL NOT NULL,
                    descricao TEXT,
                    data TEXT NOT NULL DEFAULT CURRENT_TIMESTAMP
                )
```

```
            """)
        print("BANCO DE DADOS CRIADO!")

if __name__ == '__main__':
    db_manager = DatabaseManager()
    db_manager.create_tables()
```

Explicação Linha a Linha:

- **`import sqlite3`**: Importa o módulo `sqlite3` do Python, que fornece uma interface para interagir com bancos de dados SQLite.
- **`class DatabaseManager:`**: Define a classe `DatabaseManager`, que encapsula a lógica de conexão e gerenciamento do banco de dados.
- **`def __init__(self, db_name="clientes.bd"):`**: O construtor da classe. Quando uma instância de `DatabaseManager` é criada, ele inicializa o atributo `self.db_name` com o nome do arquivo do banco de dados. Por padrão, o nome é "clientes.bd".
- **`def __enter__(self):`**: Este método é parte do protocolo de *context manager* do Python. Ele é chamado quando a instrução `with` é usada com uma instância de `DatabaseManager`.:
  - **`self.conn = sqlite3.connect(self.db_name)`**: Estabelece uma conexão com o banco de dados SQLite especificado por `self.db_name`. Se o arquivo do banco de dados não existir, ele será criado.
  - **`self.cursor = self.conn.cursor()`**: Cria um objeto *cursor* a partir da conexão. O cursor é usado para executar comandos SQL.
  - **`return self.cursor`**: Retorna o objeto cursor, que será atribuído à variável após o `as` na instrução `with` (ex: `with self as cursor:`).
- **`def __exit__(self, exc_type, exc_val, exc_tb):`**: Este método também faz parte do protocolo de *context manager*. Ele é chamado quando o bloco `with` é encerrado, independentemente de ter ocorrido uma exceção ou não.:
  - **`self.conn.commit()`**: Confirma todas as transações pendentes no banco de dados. Isso garante que as alterações feitas dentro do bloco `with` sejam salvas permanentemente.

    - `self.conn.close()`: Fecha a conexão com o banco de dados, liberando os recursos.

- `def create_tables(self):`: Este método é responsável por criar todas as tabelas necessárias no banco de dados, caso elas ainda não existam.:
    - `with self as cursor:`: Utiliza a própria instância de `DatabaseManager` como um *context manager*. Isso garante que a conexão seja aberta, as operações sejam executadas e a conexão seja fechada e confirmada automaticamente.
    - `cursor.execute(""" CREATE TABLE IF NOT EXISTS ... """)`: Para cada tabela (usuários, clientes, fornecedores, produtos, estoque, vendas, compras, financeiro), um comando SQL `CREATE TABLE IF NOT EXISTS` é executado. O `IF NOT EXISTS` é crucial, pois evita erros se a tabela já existir.
        - `id_... INTEGER PRIMARY KEY AUTOINCREMENT`: Define uma coluna de ID que é um número inteiro, chave primária e auto-incrementa automaticamente a cada nova entrada.
        - `TEXT NOT NULL`: Define uma coluna de texto que não pode ser nula (deve ter um valor).
        - `TEXT NOT NULL UNIQUE`: Define uma coluna de texto que não pode ser nula e todos os valores nesta coluna devem ser únicos (ex: CPF, Email, CNPJ).
        - `REAL NOT NULL`: Define uma coluna para números de ponto flutuante (reais) que não pode ser nula (usado para preços).
        - `INTEGER NOT NULL DEFAULT 0`: Define uma coluna de número inteiro que não pode ser nula e tem um valor padrão de 0 se nenhum valor for fornecido (usado para quantidade em estoque).
        - `CHECK(tipo IN (...))`: Uma restrição que garante que os valores na coluna `tipo` sejam apenas um dos valores especificados (ex: 'admin', 'vendedor').
        - `DEFAULT 'padrao'`: Define um valor padrão para a coluna `permissao` se nenhum for fornecido.
        - `FOREIGN KEY(...) REFERENCES ...(...)`: Define chaves estrangeiras, que criam um vínculo entre tabelas. Por exemplo,

`fornecedor_id` na tabela `produtos` referencia `id_fornecedor` na tabela `fornecedores`, garantindo a integridade referencial.

    - **`print("BANCO DE DADOS CRIADO!")`**: Uma mensagem de confirmação é impressa no console após a criação de todas as tabelas.

- **`if __name__ == '__main__':`**: Este bloco de código é executado apenas quando o script `database_manager.py` é executado diretamente (não quando é importado como um módulo em outro script).:

    - **`db_manager = DatabaseManager()`**: Cria uma instância da classe `DatabaseManager`.
    - **`db_manager.create_tables()`**: Chama o método `create_tables()` para criar as tabelas no banco de dados. Isso é útil para inicializar o banco de dados pela primeira vez ou para garantir que todas as tabelas existam durante o desenvolvimento.

Em resumo, `database_manager.py` é a fundação do sistema, responsável por configurar e manter a estrutura do banco de dados de forma robusta e segura.

2. `models/models.py`

Este arquivo define as estruturas de dados (modelos) para as diferentes entidades do sistema ERP, como usuários, clientes, fornecedores, produtos, estoque, vendas, compras e financeiro. Ele utiliza `dataclasses` do Python, que são uma forma conveniente de criar classes que servem principalmente para armazenar dados.

```python
 from dataclasses import dataclass, field
from datetime import datetime
from typing import Optional

@dataclass
class User:
    id_usuario: Optional[int] = None
    nome_usuario: str = field(default="")
    cpf_usuario: str = field(default="")
    email_usuario: str = field(default="")
    telefone_usuario: str = field(default="")
    data_nascimento: str = field(default="")
    rua: str = field(default="")
    cep: str = field(default="")
    bairro: str = field(default="")
    cidade: str = field(default="")
    senha: str = field(default="")
    tipo: str = field(default="vendedor") # admin, vendedor, financeiro,
estoque
    permissao: str = field(default="padrao")

@dataclass
class Client:
    id_cliente: Optional[int] = None
    nome_cliente: str = field(default="")
    cpf_cliente: str = field(default="")
    email_cliente: str = field(default="")
    telefone_cliente: str = field(default="")
    data_nascimento: str = field(default="")
    rua: str = field(default="")
    cep: str = field(default="")
    bairro: str = field(default="")
    cidade: str = field(default="")

@dataclass
class Supplier:
    id_fornecedor: Optional[int] = None
    nome: str = field(default="")
    cnpj: str = field(default="")
    telefone: Optional[str] = None
    email: Optional[str] = None
    rua: str = field(default="")
    cep: str = field(default="")
    bairro: str = field(default="")
    cidade: str = field(default="")

@dataclass
class Product:
    id_produto: Optional[int] = None
    nome: str = field(default="")
    descricao: Optional[str] = None
    preco_venda: float = field(default=0.0)
    preco_compra: float = field(default=0.0)
    fornecedor_id: Optional[int] = None

@dataclass
class Stock:
    id_estoque: Optional[int] = None
    produto_id: int = field(default=0)
    quantidade: int = field(default=0)
```

```python
@dataclass
class Sale:
    id_vendas: Optional[int] = None
    cliente_id: int = field(default=0)
    usuario_id: int = field(default=0)
    data_venda: str = field(default_factory=lambda:
datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
    total: float = field(default=0.0)

@dataclass
class Purchase:
    id_compras: Optional[int] = None
    fornecedor_id: int = field(default=0)
    usuario_id: int = field(default=0)
    data_compra: str = field(default_factory=lambda:
datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
    total: float = field(default=0.0)

@dataclass
class Financial:
    id_financeiro: Optional[int] = None
    tipo: str = field(default="entrada") # entrada, saida
    valor: float = field(default=0.0)
    descricao: Optional[str] = None
    data: str = field(default_factory=lambda: datetime.now().strftime("%Y-%m-%d
%H:%M:%S"))
```

Explicação Linha a Linha:

- **`from dataclasses import dataclass, field`**: Importa os decoradores `dataclass` e a função `field` do módulo `dataclasses`. `dataclass` é usado para criar classes que são principalmente contêineres para dados, e `field` permite configurar atributos de classe, como valores padrão.
- **`from datetime import datetime`**: Importa a classe `datetime` do módulo `datetime`, usada para trabalhar com datas e horas.
- **`from typing import Optional`**: Importa `Optional` do módulo `typing`, usado para indicar que um campo pode ser de um tipo específico ou `None`.
- **`@dataclass`**: Este decorador é aplicado a cada classe e automaticamente gera métodos como `__init__`, `__repr__`, `__eq__`, etc., com base nos campos definidos na classe. Isso reduz a quantidade de código *boilerplate*.
- **`id_...: Optional[int] = None`**: Em todas as classes, os campos `id_` (como `id_usuario`, `id_cliente`, etc.) são definidos como `Optional[int]` e inicializados com `None`. Isso significa que o ID pode ser um número inteiro ou `None`. Geralmente, o ID é `None` quando um novo objeto é criado (pois o banco

de dados ainda não gerou um ID para ele) e é preenchido com um `int` após ser salvo no banco de dados.

- **`nome_...: str = field(default="")`**: Campos como `nome_usuario`, `nome_cliente`, etc., são definidos como `str` e usam `field(default="")`. Isso define um valor padrão de uma string vazia (`""`) para o campo se nenhum valor for fornecido durante a criação do objeto. O uso de `field` é necessário para valores padrão mutáveis ou quando se deseja configurar metadados adicionais para o campo.

- **`cpf_usuario: str = field(default="")`**: Campos como CPF e CNPJ são strings, pois podem conter caracteres não numéricos (pontos, traços) e não são usados em cálculos matemáticos.

- **`telefone: Optional[str] = None`**: Para campos como `telefone` e `email` em `Supplier` (e `descricao` em `Product` e `Financial`), eles são `Optional[str] = None`. Isso indica que esses campos podem ser strings ou `None`, permitindo que sejam opcionais e não obrigatórios.

- **`preco_venda: float = field(default=0.0)`**: Campos numéricos como `preco_venda`, `preco_compra`, `valor` e `total` são definidos como `float` (para números decimais) e inicializados com `field(default=0.0)`.

- **`quantidade: int = field(default=0)`**: O campo `quantidade` em `Stock` é um `int` e inicializado com `field(default=0)`.

- **`fornecedor_id: Optional[int] = None`**: Este campo em `Product` é um `Optional[int]`, representando a chave estrangeira para a tabela de fornecedores. Ele pode ser `None` se um produto não estiver associado a um fornecedor específico.

- **`tipo: str = field(default="vendedor")`**: Em `User` e `Financial`, o campo `tipo` tem um valor padrão e um comentário indicando os valores permitidos. Isso ajuda a documentar as opções válidas para esses campos.

- **`data_venda: str = field(default_factory=lambda: datetime.now().strftime("%Y-%m-%d %H:%M:%S"))`**: Campos de data e hora, como `data_venda`, `data_compra` e `data` em `Financial`, usam `default_factory`. Isso é crucial para campos com valores padrão que precisam

ser gerados dinamicamente (como a data e hora atuais). `lambda: datetime.now().strftime("%Y-%m-%d %H:%M:%S")` cria uma função anônima que retorna a data e hora atuais formatadas como uma string quando o objeto é criado. Se `default=datetime.now().strftime(...)` fosse usado, a data e hora seriam definidas apenas uma vez, quando a classe fosse definida, e não quando cada nova instância fosse criada.

Em resumo, `models.py` fornece uma representação clara e tipada dos dados do sistema, facilitando a manipulação e validação das informações em todo o aplicativo. A utilização de `dataclasses` simplifica a criação desses modelos, tornando o código mais conciso e legível.

3. `business_logic/client_manager.py`

Este arquivo contém a lógica de negócios para gerenciar as operações relacionadas aos clientes no sistema ERP. Ele atua como uma camada intermediária entre a interface do usuário e o banco de dados, utilizando o `DatabaseManager` para interagir com o SQLite e o modelo `Client` para representar os dados dos clientes.

```python
 from ..database.database_manager import DatabaseManager
from ..models.models import Client

class ClientManager:
    def __init__(self):
        self.db_manager = DatabaseManager()

    def add_client(self, client: Client):
        with self.db_manager as cursor:
            cursor.execute(""" INSERT INTO clientes (nome_cliente, cpf_cliente,
email_cliente, telefone_cliente,
                                                     data_nascimento, rua,
cep, bairro, cidade)
                                                     VALUES
(?,?,?,?,?,?,?,?,?) """,
                           (client.nome_cliente, client.cpf_cliente,
client.email_cliente, client.telefone_cliente,
                            client.data_nascimento, client.rua, client.cep,
client.bairro, client.cidade))
        return True

    def get_all_clients(self):
        with self.db_manager as cursor:
            cursor.execute(""" SELECT * FROM clientes ORDER BY nome_cliente
ASC; """)
            rows = cursor.fetchall()
            return [Client(id_cliente=row[0], nome_cliente=row[1],
cpf_cliente=row[2], email_cliente=row[3],
                           telefone_cliente=row[4], data_nascimento=row[5],
rua=row[6], cep=row[7],
                           bairro=row[8], cidade=row[9]) for row in rows]

    def delete_client(self, client_id: int):
        with self.db_manager as cursor:
            cursor.execute("""DELETE FROM clientes WHERE id_cliente = ?""",
(client_id,))
        return True

    def update_client(self, client: Client):
        with self.db_manager as cursor:
            cursor.execute("""
            UPDATE clientes
            SET nome_cliente = ?, cpf_cliente = ?, email_cliente = ?,
telefone_cliente = ?, data_nascimento = ?,
             rua = ?, cep = ?, bairro = ?, cidade = ? WHERE id_cliente = ?""",
                           (client.nome_cliente, client.cpf_cliente,
client.email_cliente, client.telefone_cliente,
                            client.data_nascimento, client.rua, client.cep,
client.bairro, client.cidade, client.id_cliente))
        return True

    def search_client(self, name: str):
        with self.db_manager as cursor:
            # Prevenção de SQL Injection: usar LIKE com parâmetros
            cursor.execute("SELECT * FROM clientes WHERE nome_cliente LIKE ?
ORDER BY nome_cliente ASC", (f'%{name}%',))
            rows = cursor.fetchall()
            return [Client(id_cliente=row[0], nome_cliente=row[1],
cpf_cliente=row[2], email_cliente=row[3],
                           telefone_cliente=row[4], data_nascimento=row[5],
```

```
rua=row[6], cep=row[7],
                            bairro=row[8], cidade=row[9]) for row in rows]
```

Explicação Linha a Linha:

- **`from ..database.database_manager import DatabaseManager`**: Importa a classe `DatabaseManager` do diretório `database`. Os dois pontos (`..`) indicam que a importação é relativa ao diretório pai.
- **`from ..models.models import Client`**: Importa a classe `Client` do diretório `models`, que representa a estrutura de dados de um cliente.
- **`class ClientManager:`**: Define a classe `ClientManager`, que agrupa todas as operações de negócios relacionadas a clientes.
- **`def __init__(self):`**: O construtor da classe. Ele inicializa uma instância de `DatabaseManager`, que será usada para todas as interações com o banco de dados.
    - **`self.db_manager = DatabaseManager()`**: Cria uma instância do gerenciador de banco de dados.
- **`def add_client(self, client: Client):`**: Adiciona um novo cliente ao banco de dados.
    - **`client: Client`**: A anotação de tipo indica que o parâmetro `client` deve ser uma instância da classe `Client`.
    - **`with self.db_manager as cursor:`**: Utiliza o `DatabaseManager` como um *context manager* para obter um cursor do banco de dados. Isso garante que a conexão seja aberta, a transação seja confirmada e a conexão seja fechada automaticamente.
    - **`cursor.execute(""" INSERT INTO clientes (...) VALUES (?,?,?,?,?,?,?,?,?) """, (...))`**: Executa uma instrução SQL `INSERT` para adicionar os dados do cliente na tabela `clientes`. Os `?` são *placeholders* para os valores, e a tupla `(...)` fornece os valores correspondentes. Isso é uma prática de segurança fundamental para prevenir SQL Injection.

    - `return True`: Retorna `True` para indicar que a operação foi bem-sucedida.

- `def get_all_clients(self):`: Recupera todos os clientes do banco de dados.
    - `cursor.execute(""" SELECT * FROM clientes ORDER BY nome_cliente ASC; """)`: Executa uma consulta `SELECT` para obter todos os registros da tabela `clientes`, ordenados pelo nome do cliente.
    - `rows = cursor.fetchall()`: Recupera todas as linhas resultantes da consulta.
    - `return [Client(...) for row in rows]`: Itera sobre as linhas recuperadas e cria uma lista de objetos `Client`, mapeando os dados de cada linha para os atributos da classe `Client`.

- `def delete_client(self, client_id: int):`: Deleta um cliente do banco de dados com base no seu ID.
    - `client_id: int`: O ID do cliente a ser deletado.
    - `cursor.execute("""DELETE FROM clientes WHERE id_cliente = ?""", (client_id,))`: Executa uma instrução `DELETE` usando o ID do cliente como parâmetro seguro.

- `def update_client(self, client: Client):`: Atualiza os dados de um cliente existente no banco de dados.
    - `cursor.execute(""" UPDATE clientes SET ... WHERE id_cliente = ?""", (...))`: Executa uma instrução `UPDATE` para modificar os campos do cliente, usando o `id_cliente` para identificar o registro a ser atualizado. Todos os valores são passados como parâmetros para segurança.

- `def search_client(self, name: str):`: Busca clientes pelo nome (ou parte do nome).
    - `name: str`: O termo de busca para o nome do cliente.
    - `cursor.execute("SELECT * FROM clientes WHERE nome_cliente LIKE ? ORDER BY nome_cliente ASC", (f'%{name}%',))`: Executa uma consulta `SELECT` usando o operador `LIKE` para encontrar nomes que contenham o

termo de busca. O `f'%{name}%'` cria a string de busca com curingas (`%`) e é passado como um parâmetro separado para evitar SQL Injection, mesmo com `LIKE`.

- **`rows = cursor.fetchall()`**: Recupera as linhas que correspondem à busca.
- **`return [Client(...) for row in rows]`**: Retorna uma lista de objetos `Client` encontrados.

Em resumo, `client_manager.py` centraliza a lógica de manipulação de dados de clientes, garantindo operações seguras e consistentes com o banco de dados, e facilitando a integração com a interface do usuário ou outras partes do sistema.

4. `business_logic/user_manager.py`

Este arquivo gerencia as operações relacionadas aos usuários do sistema, incluindo a crucial funcionalidade de hashing de senhas para segurança. Ele interage com o `DatabaseManager` para persistir os dados e com o modelo `User` para representar as informações dos usuários.

```python
from ..database.database_manager import DatabaseManager
from ..models.models import User
import bcrypt

class UserManager:
    def __init__(self):
        self.db_manager = DatabaseManager()

    def hash_password(self, password):
        # Hash a password for the first time, with a randomly generated salt
        return bcrypt.hashpw(password.encode("utf-8"),
bcrypt.gensalt()).decode("utf-8")

    def check_password(self, password, hashed_password):
        # Check if the provided password matches the stored hash
        return bcrypt.checkpw(password.encode("utf-8"),
hashed_password.encode("utf-8"))

    def add_user(self, user: User):
        hashed_pw = self.hash_password(user.senha)
        with self.db_manager as cursor:
            cursor.execute(""" INSERT INTO usuarios (nome_usuario, cpf_usuario,
email_usuario, telefone_usuario,
                                                    data_nascimento, rua,
cep, bairro, cidade, senha, tipo, permissao)
                                                    VALUES
(?,?,?,?,?,?,?,?,?,?,?,?) """,
                               (user.nome_usuario, user.cpf_usuario,
user.email_usuario, user.telefone_usuario,
                                user.data_nascimento, user.rua, user.cep,
user.bairro, user.cidade, hashed_pw, user.tipo, user.permissao))
        return True

    def get_all_users(self):
        with self.db_manager as cursor:
            cursor.execute(""" SELECT * FROM usuarios ORDER BY nome_usuario
ASC; """)
            rows = cursor.fetchall()
            return [User(id_usuario=row[0], nome_usuario=row[1],
cpf_usuario=row[2], email_usuario=row[3],
                         telefone_usuario=row[4], data_nascimento=row[5],
rua=row[6], cep=row[7],
                         bairro=row[8], cidade=row[9], senha=row[10],
tipo=row[11], permissao=row[12]) for row in rows]

    def delete_user(self, user_id: int):
        with self.db_manager as cursor:
            cursor.execute("""DELETE FROM usuarios WHERE id_usuario = ?""",
(user_id,))
        return True

    def update_user(self, user: User):
        # Only hash password if it's a new password (not already hashed)
        current_user = self.get_user_by_id(user.id_usuario)
        if current_user and not self.check_password(user.senha,
current_user.senha):
            hashed_pw = self.hash_password(user.senha)
        else:
            hashed_pw = user.senha # Password is either empty or already hashed

        with self.db_manager as cursor:
```

```
            cursor.execute("""
            UPDATE usuarios
            SET nome_usuario = ?, cpf_usuario = ?, email_usuario = ?,
telefone_usuario = ?, data_nascimento = ?,
             rua = ?, cep = ?, bairro = ?, cidade = ?, senha = ?, tipo = ?,
permissao = ? WHERE id_usuario = ?""",
                           (user.nome_usuario, user.cpf_usuario,
user.email_usuario, user.telefone_usuario,
                            user.data_nascimento, user.rua, user.cep,
user.bairro, user.cidade, hashed_pw, user.tipo, user.permissao,
user.id_usuario))
        return True

    def search_user(self, name: str):
        with self.db_manager as cursor:
            cursor.execute("SELECT * FROM usuarios WHERE nome_usuario LIKE ?
ORDER BY nome_usuario ASC", (f'%{name}%',))
            rows = cursor.fetchall()
            return [User(id_usuario=row[0], nome_usuario=row[1],
cpf_usuario=row[2], email_usuario=row[3],
                         telefone_usuario=row[4], data_nascimento=row[5],
rua=row[6], cep=row[7],
                         bairro=row[8], cidade=row[9], senha=row[10],
tipo=row[11], permissao=row[12]) for row in rows]

    def get_user_by_id(self, user_id: int):
        with self.db_manager as cursor:
            cursor.execute("SELECT * FROM usuarios WHERE id_usuario = ?",
(user_id,))
            row = cursor.fetchone()
            if row:
                return User(id_usuario=row[0], nome_usuario=row[1],
cpf_usuario=row[2], email_usuario=row[3],
                            telefone_usuario=row[4], data_nascimento=row[5],
rua=row[6], cep=row[7],
                            bairro=row[8], cidade=row[9], senha=row[10],
tipo=row[11], permissao=row[12])
            return None
```

Explicação Linha a Linha:

- `from ..database.database_manager import DatabaseManager`: Importa a classe `DatabaseManager` para interagir com o banco de dados.
- `from ..models.models import User`: Importa a classe `User` que define a estrutura de dados para um usuário.
- `import bcrypt`: Importa a biblioteca `bcrypt`, utilizada para realizar o hashing seguro de senhas.
- `class UserManager:`: Define a classe `UserManager`, que encapsula a lógica de negócios para operações de usuário.

- `def __init__(self):`: O construtor da classe, que inicializa uma instância de `DatabaseManager`.
- `def hash_password(self, password):`: Este método recebe uma senha em texto simples e retorna seu hash seguro.
    - `password.encode("utf-8")`: Converte a senha de string para bytes, que é o formato exigido pelo `bcrypt`.
    - `bcrypt.gensalt()`: Gera um *salt* aleatório. O *salt* é um dado aleatório que é combinado com a senha antes do hashing, tornando cada hash único, mesmo para senhas idênticas. Isso protege contra ataques de tabela arco-íris.
    - `bcrypt.hashpw(..., ...)`: Realiza o hashing da senha usando o algoritmo bcrypt e o *salt* gerado.
    - `.decode("utf-8")`: Converte o hash resultante de bytes de volta para uma string para armazenamento.
- `def check_password(self, password, hashed_password):`: Este método verifica se uma senha fornecida corresponde a um hash armazenado.
    - `password.encode("utf-8")`: Converte a senha fornecida para bytes.
    - `hashed_password.encode("utf-8")`: Converte o hash armazenado para bytes.
    - `bcrypt.checkpw(..., ...)`: Compara a senha fornecida (após ser hashed com o *salt* do `hashed_password`) com o `hashed_password`. Retorna `True` se elas corresponderem, `False` caso contrário.
- `def add_user(self, user: User):`: Adiciona um novo usuário ao banco de dados.
    - `hashed_pw = self.hash_password(user.senha)`: Antes de inserir o usuário, a senha é hashed usando o método `hash_password`.
    - `cursor.execute(""" INSERT INTO usuarios (...) VALUES (?,?,?,?,?,?,?,?,?,?,?,?) """, (...))`: Insere os dados do usuário na tabela `usuarios`. O `hashed_pw` é armazenado em vez da senha em texto simples.

- `def get_all_users(self):`: Recupera todos os usuários do banco de dados.
    - `cursor.execute(""" SELECT * FROM usuarios ORDER BY nome_usuario ASC; """)`: Seleciona todos os usuários, ordenados pelo nome.
    - `return [User(...) for row in rows]`: Mapeia as linhas do banco de dados para objetos `User`.
- `def delete_user(self, user_id: int):`: Deleta um usuário pelo seu ID.
- `def update_user(self, user: User):`: Atualiza os dados de um usuário existente.
    - `current_user = self.get_user_by_id(user.id_usuario)`: Recupera os dados atuais do usuário do banco de dados para verificar a senha.
    - `if current_user and not self.check_password(user.senha, current_user.senha):`: Verifica se a senha fornecida no objeto `user` é diferente da senha hashed atualmente armazenada para `current_user`. Se for diferente, significa que uma nova senha foi fornecida e precisa ser hashed.
    - `hashed_pw = self.hash_password(user.senha)`: Hashes a nova senha.
    - `else: hashed_pw = user.senha`: Se a senha não mudou (ou se o campo de senha foi deixado vazio na GUI), o hash existente é mantido. Isso evita re-hashing desnecessário ou a perda do hash se o campo de senha não for preenchido na atualização.
    - `cursor.execute(""" UPDATE usuarios SET ... WHERE id_usuario = ?""", (...))`: Atualiza os dados do usuário no banco de dados, incluindo o `hashed_pw`.
- `def search_user(self, name: str):`: Busca usuários pelo nome, de forma similar ao `search_client`.
- `def get_user_by_id(self, user_id: int):`: Recupera um único usuário pelo seu ID. Este método é útil internamente para verificar a senha atual antes de uma atualização.

Em resumo, `user_manager.py` é fundamental para o gerenciamento de usuários, com um foco especial na segurança das senhas através do hashing com `bcrypt`,

garantindo que as credenciais dos usuários sejam protegidas adequadamente.

5. `business_logic/supplier_manager.py`

Este arquivo é dedicado à gestão dos fornecedores no sistema ERP. Ele fornece métodos para adicionar, recuperar, atualizar e deletar informações de fornecedores, interagindo com o banco de dados através do `DatabaseManager` e utilizando o modelo `Supplier` para a representação dos dados.

```python
from ..database.database_manager import DatabaseManager
from ..models.models import Supplier

class SupplierManager:
    def __init__(self):
        self.db_manager = DatabaseManager()

    def add_supplier(self, supplier: Supplier):
        with self.db_manager as cursor:
            cursor.execute(""" INSERT INTO fornecedores (nome, cnpj, telefone,
email, rua, cep, bairro, cidade)
                                                    VALUES (?,?,?,?,?,?,?,?)
""",
                           (supplier.nome, supplier.cnpj, supplier.telefone,
supplier.email, supplier.rua,
                            supplier.cep, supplier.bairro, supplier.cidade))
        return True

    def get_all_suppliers(self):
        with self.db_manager as cursor:
            cursor.execute(""" SELECT * FROM fornecedores ORDER BY nome ASC;
""")
            rows = cursor.fetchall()
            return [Supplier(id_fornecedor=row[0], nome=row[1], cnpj=row[2],
telefone=row[3], email=row[4],
                             rua=row[5], cep=row[6], bairro=row[7],
cidade=row[8]) for row in rows]

    def delete_supplier(self, supplier_id: int):
        with self.db_manager as cursor:
            cursor.execute("""DELETE FROM fornecedores WHERE id_fornecedor =
?""", (supplier_id,))
        return True

    def update_supplier(self, supplier: Supplier):
        with self.db_manager as cursor:
            cursor.execute("""
            UPDATE fornecedores
            SET nome = ?, cnpj = ?, telefone = ?, email = ?, rua = ?, cep = ?,
bairro = ?, cidade = ? WHERE id_fornecedor = ?""",
                           (supplier.nome, supplier.cnpj, supplier.telefone,
supplier.email, supplier.rua,
                            supplier.cep, supplier.bairro, supplier.cidade,
supplier.id_fornecedor))
        return True

    def search_supplier(self, name: str):
        with self.db_manager as cursor:
            cursor.execute("SELECT * FROM fornecedores WHERE nome LIKE ? ORDER
BY nome ASC", (f'%{name}%',))
            rows = cursor.fetchall()
            return [Supplier(id_fornecedor=row[0], nome=row[1], cnpj=row[2],
telefone=row[3], email=row[4],
                             rua=row[5], cep=row[6], bairro=row[7],
cidade=row[8]) for row in rows]
```

Explicação Linha a Linha:

- `from ..database.database_manager import DatabaseManager`: Importa a classe `DatabaseManager` para gerenciar a conexão e as operações com o banco de dados SQLite.
- `from ..models.models import Supplier`: Importa a classe `Supplier` do módulo `models`, que define a estrutura de dados para um fornecedor.
- `class SupplierManager:`: Define a classe `SupplierManager`, que é responsável por toda a lógica de negócios relacionada aos fornecedores.
- `def __init__(self):`: O construtor da classe. Ele inicializa uma instância de `DatabaseManager`, que será usada por todos os métodos desta classe para interagir com o banco de dados.
    - `self.db_manager = DatabaseManager()`: Cria uma instância do gerenciador de banco de dados.
- `def add_supplier(self, supplier: Supplier):`: Adiciona um novo fornecedor ao banco de dados.
    - `supplier: Supplier`: O parâmetro `supplier` é uma instância da classe `Supplier`, contendo os dados do novo fornecedor.
    - `with self.db_manager as cursor:`: Abre uma conexão com o banco de dados usando o `DatabaseManager` como um *context manager*, garantindo que a conexão seja gerenciada corretamente.
    - `cursor.execute(""" INSERT INTO fornecedores (nome, cnpj, telefone, email, rua, cep, bairro, cidade) VALUES (?,?,?,?,?,?,?,?) """, (...))`: Executa uma instrução SQL `INSERT` para inserir os dados do fornecedor na tabela `fornecedores`. Os `?` são *placeholders* para os valores, e a tupla `(...)` fornece os atributos do objeto `supplier` na ordem correta. Isso é crucial para prevenir ataques de SQL Injection.
    - `return True`: Retorna `True` indicando que a operação de adição foi bem-sucedida.

- `def get_all_suppliers(self):`: Recupera todos os fornecedores registrados no banco de dados.
    - `cursor.execute(""" SELECT * FROM fornecedores ORDER BY nome ASC; """)`: Executa uma consulta `SELECT` para obter todas as colunas de todos os registros na tabela `fornecedores`, ordenados alfabeticamente pelo nome.
    - `rows = cursor.fetchall()`: Obtém todas as linhas retornadas pela consulta como uma lista de tuplas.
    - `return [Supplier(...) for row in rows]`: Itera sobre cada tupla (`row`) na lista `rows` e cria uma nova instância da classe `Supplier` para cada uma, mapeando os valores das colunas do banco de dados para os atributos do objeto `Supplier`. Retorna uma lista desses objetos `Supplier`.
- `def delete_supplier(self, supplier_id: int):`: Deleta um fornecedor específico do banco de dados com base no seu ID.
    - `supplier_id: int`: O ID do fornecedor a ser deletado.
    - `cursor.execute("""DELETE FROM fornecedores WHERE id_fornecedor = ?""", (supplier_id,))`: Executa uma instrução SQL `DELETE` para remover o registro correspondente ao `supplier_id` fornecido. O `(supplier_id,)` é uma tupla de um único elemento, necessária para o `execute`.
    - `return True`: Retorna `True` indicando que a operação de exclusão foi bem-sucedida.
- `def update_supplier(self, supplier: Supplier):`: Atualiza as informações de um fornecedor existente no banco de dados.
    - `supplier: Supplier`: O objeto `Supplier` contendo os dados atualizados do fornecedor, incluindo seu `id_fornecedor`.
    - `cursor.execute(""" UPDATE fornecedores SET ... WHERE id_fornecedor = ?""", (...))`: Executa uma instrução SQL `UPDATE` para modificar os campos do fornecedor. Os valores são passados como parâmetros, e a cláusula `WHERE` usa o `id_fornecedor` do objeto `supplier` para identificar qual registro deve ser atualizado.

    - **`return True`**: Retorna `True` indicando que a operação de atualização foi bem-sucedida.

- **`def search_supplier(self, name: str):`**: Busca fornecedores pelo nome ou parte do nome.
    - **`name: str`**: O termo de busca para o nome do fornecedor.
    - **`cursor.execute("SELECT * FROM fornecedores WHERE nome LIKE ? ORDER BY nome ASC", (f'%{name}%',))`**: Executa uma consulta `SELECT` que usa o operador `LIKE` para encontrar fornecedores cujo nome contenha o termo de busca (`%name%`). O `f'%{name}%'` é uma f-string que insere o termo de busca com curingas, e a tupla `(f'%{name}%',)` garante que o termo seja passado como um parâmetro seguro, prevenindo SQL Injection.
    - **`rows = cursor.fetchall()`**: Obtém as linhas que correspondem à busca.
    - **`return [Supplier(...) for row in rows]`**: Retorna uma lista de objetos `Supplier` criados a partir das linhas encontradas.

Em suma, `supplier_manager.py` centraliza e padroniza todas as interações com os dados de fornecedores, garantindo a integridade e a segurança das informações através do uso de consultas parametrizadas e do `DatabaseManager`.

6. `business_logic/product_manager.py`

Este arquivo é responsável por gerenciar os produtos e o estoque no sistema ERP. Ele lida com as operações CRUD para produtos e também com a atualização da quantidade em estoque, interagindo com o `DatabaseManager` e utilizando os modelos `Product` e `Stock`.

```python
from ..database.database_manager import DatabaseManager
from ..models.models import Product, Stock

class ProductManager:
    def __init__(self):
        self.db_manager = DatabaseManager()

    def add_product(self, product: Product, initial_stock: int = 0):
        with self.db_manager as cursor:
            cursor.execute(""" INSERT INTO produtos (nome, descricao,
preco_venda, preco_compra, fornecedor_id)
                                                    VALUES (?,?,?,?,?) """,
                           (product.nome, product.descricao,
product.preco_venda, product.preco_compra, product.fornecedor_id))
            product_id = cursor.lastrowid
            if product_id and initial_stock > 0:
                cursor.execute(""" INSERT INTO estoque (produto_id, quantidade)
VALUES (?,?) """, (product_id, initial_stock))
        return True

    def get_all_products(self):
        with self.db_manager as cursor:
            cursor.execute(""" SELECT p.*, s.quantidade FROM produtos p LEFT
JOIN estoque s ON p.id_produto = s.produto_id ORDER BY p.nome ASC; """)
            rows = cursor.fetchall()
            products = []
            for row in rows:
                product = Product(id_produto=row[0], nome=row[1],
descricao=row[2], preco_venda=row[3],
                                  preco_compra=row[4], fornecedor_id=row[5])
                product.stock_quantity = row[6] if row[6] is not None else 0 #
Adiciona a quantidade em estoque
                products.append(product)
            return products

    def delete_product(self, product_id: int):
        with self.db_manager as cursor:
            cursor.execute("""DELETE FROM estoque WHERE produto_id = ?""",
(product_id,))
            cursor.execute("""DELETE FROM produtos WHERE id_produto = ?""",
(product_id,))
        return True

    def update_product(self, product: Product):
        with self.db_manager as cursor:
            cursor.execute("""
            UPDATE produtos
            SET nome = ?, descricao = ?, preco_venda = ?, preco_compra = ?,
fornecedor_id = ? WHERE id_produto = ?""",
                           (product.nome, product.descricao,
product.preco_venda, product.preco_compra, product.fornecedor_id,
product.id_produto))
        return True

    def update_stock(self, product_id: int, quantity: int):
        with self.db_manager as cursor:
            cursor.execute(""" INSERT OR REPLACE INTO estoque (produto_id,
quantidade) VALUES (?, (SELECT COALESCE(quantidade, 0) FROM estoque WHERE
produto_id = ?) + ?) """,
                           (product_id, product_id, quantity))
        return True
```

```
    def search_product(self, name: str):
        with self.db_manager as cursor:
            cursor.execute("SELECT p.*, s.quantidade FROM produtos p LEFT JOIN
estoque s ON p.id_produto = s.produto_id WHERE p.nome LIKE ? ORDER BY p.nome
ASC", (f'%{name}%',))
            rows = cursor.fetchall()
            products = []
            for row in rows:
                product = Product(id_produto=row[0], nome=row[1],
descricao=row[2], preco_venda=row[3],
                                  preco_compra=row[4], fornecedor_id=row[5])
                product.stock_quantity = row[6] if row[6] is not None else 0
                products.append(product)
            return products
```

Explicação Linha a Linha:

- **`from ..database.database_manager import DatabaseManager`**: Importa a classe `DatabaseManager` para gerenciar a conexão e as operações com o banco de dados.
- **`from ..models.models import Product, Stock`**: Importa as classes `Product` e `Stock` do módulo `models`, que definem as estruturas de dados para produtos e seus respectivos estoques.
- **`class ProductManager:`**: Define a classe `ProductManager`, que encapsula toda a lógica de negócios para produtos e estoque.
- **`def __init__(self):`**: O construtor da classe, que inicializa uma instância de `DatabaseManager` para interagir com o banco de dados.
- **`def add_product(self, product: Product, initial_stock: int = 0):`**: Adiciona um novo produto ao banco de dados e, opcionalmente, uma quantidade inicial ao estoque.
    - **`product: Product`**: Uma instância da classe `Product` com os dados do novo produto.
    - **`initial_stock: int = 0`**: A quantidade inicial de estoque para este produto, com valor padrão de 0.
    - **`cursor.execute(""" INSERT INTO produtos (...) VALUES (?,?,?,?,?) """, (...))`**: Insere os dados do produto na tabela `produtos`.

    - `product_id = cursor.lastrowid`: Obtém o ID gerado automaticamente para o produto recém-inserido. Este ID é crucial para associar o estoque ao produto.
    - `if product_id and initial_stock > 0:`: Verifica se o produto foi inserido com sucesso e se há uma quantidade inicial de estoque a ser adicionada.
    - `cursor.execute(""" INSERT INTO estoque (produto_id, quantidade) VALUES (?,?) """, (product_id, initial_stock))`: Insere a quantidade inicial de estoque na tabela `estoque`, vinculando-a ao `product_id`.

- `def get_all_products(self):`: Recupera todos os produtos do banco de dados, incluindo suas quantidades em estoque.
    - `cursor.execute(""" SELECT p.*, s.quantidade FROM produtos p LEFT JOIN estoque s ON p.id_produto = s.produto_id ORDER BY p.nome ASC; """)`: Executa uma consulta `SELECT` que utiliza um `LEFT JOIN` para combinar os dados da tabela `produtos` com a tabela `estoque`. Isso garante que todos os produtos sejam retornados, mesmo aqueles que não possuem um registro correspondente na tabela `estoque` (nesse caso, `s.quantidade` será `NULL`).
    - `rows = cursor.fetchall()`: Obtém todas as linhas resultantes da consulta.
    - `for row in rows:`: Itera sobre cada linha retornada.
    - `product = Product(...)`: Cria uma instância de `Product` com os dados do produto.
    - `product.stock_quantity = row[6] if row[6] is not None else 0`: Adiciona um atributo `stock_quantity` ao objeto `product`. Se a quantidade do estoque (`row[6]`) for `None` (porque não há registro em `estoque` para aquele produto), define como 0; caso contrário, usa o valor do banco de dados.
    - `products.append(product)`: Adiciona o produto (com a quantidade de estoque) à lista de produtos.

- `def delete_product(self, product_id: int):`: Deleta um produto e seu registro de estoque associado.

    - `cursor.execute("""DELETE FROM estoque WHERE produto_id = ?""", (product_id,))`: Primeiro, deleta o registro de estoque para o produto. Isso é importante para manter a integridade referencial e evitar erros de chave estrangeira.
    - `cursor.execute("""DELETE FROM produtos WHERE id_produto = ?""", (product_id,))`: Em seguida, deleta o produto da tabela `produtos`.

- `def update_product(self, product: Product):`: Atualiza os dados de um produto existente.
    - `cursor.execute(""" UPDATE produtos SET ... WHERE id_produto = ?""", (...))`: Executa uma instrução `UPDATE` para modificar os campos do produto, usando o `id_produto` para identificar o registro.
- `def update_stock(self, product_id: int, quantity: int):`: Atualiza a quantidade de estoque de um produto. Esta função é flexível, permitindo adicionar ou remover estoque.
    - `cursor.execute(""" INSERT OR REPLACE INTO estoque (produto_id, quantidade) VALUES (?, (SELECT COALESCE(quantidade, 0) FROM estoque WHERE produto_id = ?) + ?) """, (...))`: Esta é uma consulta SQL inteligente:
        - `INSERT OR REPLACE`: Se um registro para `produto_id` já existir na tabela `estoque`, ele será atualizado; caso contrário, um novo registro será inserido.
        - `SELECT COALESCE(quantidade, 0) FROM estoque WHERE produto_id = ?`: Recupera a quantidade atual do produto no estoque. `COALESCE` garante que, se não houver registro (e `quantidade` for `NULL`), ele use 0 em vez de `NULL`.
        - `+ ?`: Adiciona a `quantity` fornecida ao valor atual. Se `quantity` for positiva, o estoque aumenta; se for negativa, o estoque diminui.
- `def search_product(self, name: str):`: Busca produtos pelo nome, de forma similar aos outros gerentes, incluindo a quantidade em estoque no resultado.

Em resumo, `product_manager.py` é crucial para o controle de inventário, permitindo um gerenciamento eficiente dos produtos e suas quantidades em estoque, com operações seguras e integradas ao banco de dados.

7. `gui/gui_components.py`

Este arquivo contém uma coleção de funções estáticas e utilitários para a interface gráfica do usuário (GUI) construída com Tkinter. O objetivo é centralizar funcionalidades comuns, como limpeza de campos, exibição de mensagens e validação de entrada, promovendo a reutilização de código e a consistência visual e funcional da aplicação.

```python
 from tkinter import Entry, END, messagebox
from tkinter.ttk import Combobox

class GUIComponents:
    @staticmethod
    def clear_entries(*entries):
        for entry in entries:
            if isinstance(entry, Entry):
                entry.delete(0, END)
            elif isinstance(entry, Combobox):
                entry.set("") # Clear combobox selection

    @staticmethod
    def show_info(title, message):
        messagebox.showinfo(title, message)

    @staticmethod
    def show_error(title, message):
        messagebox.showerror(title, message)

    @staticmethod
    def show_warning(title, message):
        messagebox.showwarning(title, message)

    @staticmethod
    def ask_yes_no(title, message):
        return messagebox.askyesno(title, message)

    @staticmethod
    def validate_not_empty(value, field_name):
        if not value.strip():
            GUIComponents.show_warning("Campo Vazio", f"O campo
\'{field_name}\' não pode estar vazio.")
            return False
        return True

    @staticmethod
    def validate_email(email):
        import re
        if not re.match(r"[^@]+@[^@]+\\.[^@]+", email):
            GUIComponents.show_warning("Email Inválido", "Por favor, insira um
endereço de e-mail válido.")
            return False
        return True

    @staticmethod
    def validate_cpf(cpf):
        # Simplistic CPF validation for demonstration. A real one would be more
complex.
        if not cpf.isdigit() or len(cpf) != 11:
            GUIComponents.show_warning("CPF Inválido", "O CPF deve conter 11
dígitos numéricos.")
            return False
        return True

    @staticmethod
    def validate_cnpj(cnpj):
        # Simplistic CNPJ validation for demonstration. A real one would be
more complex.
        if not cnpj.isdigit() or len(cnpj) != 14:
            GUIComponents.show_warning("CNPJ Inválido", "O CNPJ deve conter 14
```

```
dígitos numéricos.")
            return False
        return True

    @staticmethod
    def validate_phone(phone):
        # Allows digits, spaces, hyphens, and parentheses
        import re
        if not re.match(r"^[0-9\\s\\-()]+$", phone):
            GUIComponents.show_warning("Telefone Inválido", "O telefone contém
caracteres inválidos.")
            return False
        return True

    @staticmethod
    def validate_date(date_str):
        from datetime import datetime
        try:
            datetime.strptime(date_str, "%d/%m/%Y") # Assuming DD/MM/YYYY
format
            return True
        except ValueError:
            GUIComponents.show_warning("Data Inválida", "Formato de data
inválido. Use DD/MM/YYYY.")
            return False
```

Explicação Linha a Linha:

- **`from tkinter import Entry, END, messagebox`**: Importa classes e módulos necessários do Tkinter:
    - `Entry`: Widget de entrada de texto.
    - `END`: Constante usada para indicar o final de um texto em widgets de entrada.
    - `messagebox`: Módulo para exibir caixas de diálogo de mensagens (informação, erro, aviso, etc.).
- **`from tkinter.ttk import Combobox`**: Importa o widget `Combobox` do submódulo `ttk` do Tkinter, que oferece widgets mais modernos e com melhor aparência.
- **`class GUIComponents:`**: Define a classe `GUIComponents`, que agrupa métodos estáticos (utilitários) relacionados à GUI. Métodos estáticos não precisam de uma instância da classe para serem chamados, o que os torna ideais para funções utilitárias.

- `@staticmethod`: Decorador que indica que o método é um método estático. Ele não recebe `self` como primeiro argumento e pode ser chamado diretamente na classe (ex: `GUIComponents.clear_entries()`).
- `def clear_entries(*entries):`: Limpa o conteúdo de um ou mais widgets de entrada (Entry ou Combobox).
    - `*entries`: Permite que a função receba um número variável de argumentos (os widgets de entrada a serem limpos).
    - `for entry in entries:`: Itera sobre cada widget fornecido.
    - `if isinstance(entry, Entry):`: Verifica se o widget é uma instância de `Entry`.
        - `entry.delete(0, END)`: Deleta o texto do início (`0`) ao fim (`END`) do widget `Entry`.
    - `elif isinstance(entry, Combobox):`: Verifica se o widget é uma instância de `Combobox`.
        - `entry.set("")`: Limpa a seleção do `Combobox` definindo seu valor como uma string vazia.
- `def show_info(title, message):`: Exibe uma caixa de diálogo de informação.
    - `messagebox.showinfo(title, message)`: Chama a função `showinfo` do `messagebox` com o título e a mensagem fornecidos.
- `def show_error(title, message):`: Exibe uma caixa de diálogo de erro.
    - `messagebox.showerror(title, message)`: Chama a função `showerror` do `messagebox`.
- `def show_warning(title, message):`: Exibe uma caixa de diálogo de aviso.
    - `messagebox.showwarning(title, message)`: Chama a função `showwarning` do `messagebox`.
- `def ask_yes_no(title, message):`: Exibe uma caixa de diálogo de pergunta com opções "Sim" e "Não".
    - `messagebox.askyesno(title, message)`: Retorna `True` se o usuário clicar em "Sim" e `False` se clicar em "Não".

- `def validate_not_empty(value, field_name):`: Valida se um campo de texto não está vazio ou contém apenas espaços em branco.
    - `value.strip()`: Remove espaços em branco do início e do fim da string `value`.
    - `if not value.strip():`: Se a string resultante estiver vazia, significa que o campo está vazio.
    - `GUIComponents.show_warning(...)`: Exibe um aviso informando qual campo está vazio.
    - `return False`: Retorna `False` indicando que a validação falhou.
    - `return True`: Retorna `True` se a validação for bem-sucedida.
- `def validate_email(email):`: Valida se uma string tem o formato básico de um endereço de e-mail.
    - `import re`: Importa o módulo `re` para expressões regulares.
    - `re.match(r"[^@]+@[^@]+\\.[^@]+", email)`: Usa uma expressão regular para verificar o padrão de e-mail. Esta é uma validação simplificada; validações de e-mail mais robustas são mais complexas.
- `def validate_cpf(cpf):`: Valida se uma string é um CPF numérico de 11 dígitos. Esta é uma validação *simplista* e não inclui os algoritmos de validação de dígitos verificadores do CPF real.
    - `cpf.isdigit()`: Verifica se a string contém apenas dígitos.
    - `len(cpf) != 11`: Verifica se o comprimento é 11.
- `def validate_cnpj(cnpj):`: Valida se uma string é um CNPJ numérico de 14 dígitos. Similar à validação de CPF, esta é uma validação *simplista*.
- `def validate_phone(phone):`: Valida se uma string de telefone contém apenas dígitos, espaços, hífens e parênteses.
    - `re.match(r"^[0-9\\s\\-()]+$", phone)`: Expressão regular que permite esses caracteres.
- `def validate_date(date_str):`: Valida se uma string representa uma data no formato DD/MM/YYYY.

- **`from datetime import datetime`**: Importa `datetime` dentro da função para evitar importações desnecessárias se a função não for usada.
- **`try...except ValueError`**: Tenta converter a string `date_str` para um objeto `datetime` usando o formato `"%d/%m/%Y"`. Se a conversão falhar (por exemplo, formato incorreto ou data inválida), um `ValueError` é capturado, e um aviso é exibido.

Em resumo, `gui_components.py` é uma biblioteca de utilitários que ajuda a construir interfaces Tkinter mais limpas, com validações de entrada e feedback consistente para o usuário, reduzindo a duplicação de código em `main_app.py`.

8. `gui/main_app.py`

Este é o arquivo principal da aplicação, responsável por construir a interface gráfica do usuário (GUI) usando Tkinter e integrar todas as camadas de lógica de negócios (gerenciadores de clientes, usuários, produtos, etc.) e o gerenciador de banco de dados. Ele define a estrutura das abas, os campos de entrada, botões, tabelas (Treeview) e a lógica de interação com o usuário.

```
 import tkinter
from tkinter import *
from tkinter import ttk
from tkcalendar import DateEntry

from ..database.database_manager import DatabaseManager
from ..business_logic.client_manager import ClientManager
from ..business_logic.user_manager import UserManager
from ..business_logic.supplier_manager import SupplierManager
from ..business_logic.product_manager import ProductManager
from ..models.models import Client, User, Supplier, Product
from .gui_components import GUIComponents

class Application:
    def __init__(self):
        self.root = Tk()
        self.db_manager = DatabaseManager()
        self.client_manager = ClientManager()
        self.user_manager = UserManager()
        self.supplier_manager = SupplierManager()
        self.product_manager = ProductManager()
        self.setup_gui()
        self.root.mainloop()

    def setup_gui(self):
        self.root.title("Sistema ERP")
        self.root.geometry("800x600")

        self.notebook = ttk.Notebook(self.root)
        self.notebook.pack(pady=10, expand=True, fill="both")

        # Frames for each section
        self.client_frame = Frame(self.notebook)
        self.user_frame = Frame(self.notebook)
        self.supplier_frame = Frame(self.notebook)
        self.product_frame = Frame(self.notebook)
        self.sale_frame = Frame(self.notebook)

        self.notebook.add(self.client_frame, text="Clientes")
        self.notebook.add(self.user_frame, text="Usuários")
        self.notebook.add(self.supplier_frame, text="Fornecedores")
        self.notebook.add(self.product_frame, text="Produtos")
        self.notebook.add(self.sale_frame, text="Vendas")

        self.create_client_tab()
        self.create_user_tab()
        self.create_supplier_tab()
        self.create_product_tab()
        self.create_sale_tab()
```

Explicação Linha a Linha (Parte 1: Estrutura Principal):

- **`import tkinter`** e **`from tkinter import *`**: Importam o módulo `tkinter` e todos os seus componentes diretamente. `tkinter` é a biblioteca padrão do Python para criar interfaces gráficas.

- **`from tkinter import ttk`**: Importa o submódulo `ttk` do Tkinter, que fornece widgets mais modernos e com melhor aparência (Themed Tkinter).
- **`from tkcalendar import DateEntry`**: Importa o widget `DateEntry` da biblioteca `tkcalendar`, que permite a seleção de datas através de um calendário pop-up.
- **`from ..database.database_manager import DatabaseManager`**: Importa a classe `DatabaseManager` para gerenciar o banco de dados.
- **`from ..business_logic.client_manager import ClientManager`**: Importa a classe `ClientManager` para lidar com a lógica de clientes.
- **`from ..business_logic.user_manager import UserManager`**: Importa a classe `UserManager` para lidar com a lógica de usuários.
- **`from ..business_logic.supplier_manager import SupplierManager`**: Importa a classe `SupplierManager` para lidar com a lógica de fornecedores.
- **`from ..business_logic.product_manager import ProductManager`**: Importa a classe `ProductManager` para lidar com a lógica de produtos e estoque.
- **`from ..models.models import Client, User, Supplier, Product`**: Importa as classes de modelo de dados para as entidades principais.
- **`from .gui_components import GUIComponents`**: Importa a classe `GUIComponents` que contém funções utilitárias para a GUI.
- **`class Application:`**: Define a classe principal da aplicação Tkinter.
- **`def __init__(self):`**: O construtor da classe `Application`.
    - **`self.root = Tk()`**: Cria a janela principal da aplicação Tkinter. `self.root` é a janela raiz.
    - **`self.db_manager = DatabaseManager()`**: Cria uma instância do `DatabaseManager`. Embora as classes de gerenciamento de negócios usem seu próprio `DatabaseManager`, esta instância pode ser usada para operações globais ou de inicialização, como `create_tables`.
    - **`self.client_manager = ClientManager()`**: Cria uma instância do `ClientManager` para gerenciar as operações de clientes.

- `self.user_manager = UserManager()`: Cria uma instância do `UserManager` para gerenciar as operações de usuários.
- `self.supplier_manager = SupplierManager()`: Cria uma instância do `SupplierManager` para gerenciar as operações de fornecedores.
- `self.product_manager = ProductManager()`: Cria uma instância do `ProductManager` para gerenciar as operações de produtos.
- `self.setup_gui()`: Chama o método para configurar a interface gráfica.
- `self.root.mainloop()`: Inicia o loop principal de eventos do Tkinter. Esta linha faz com que a janela permaneça aberta e responsiva a interações do usuário. Deve ser a última linha no construtor.

- `def setup_gui(self):`: Configura a aparência e a estrutura básica da janela principal.
  - `self.root.title("Sistema ERP")`: Define o título da janela principal.
  - `self.root.geometry("800x600")`: Define o tamanho inicial da janela para 800 pixels de largura por 600 pixels de altura.
  - `self.notebook = ttk.Notebook(self.root)`: Cria um widget `Notebook` (abas) que será o contêiner principal para as diferentes seções do ERP.
  - `self.notebook.pack(pady=10, expand=True, fill="both")`: Empacota o `notebook` na janela principal. `pady=10` adiciona um preenchimento vertical, `expand=True` faz com que ele ocupe todo o espaço disponível, e `fill="both"` o expande tanto horizontal quanto verticalmente.
  - `self.client_frame = Frame(self.notebook)`: Cria um `Frame` (um contêiner retangular) para cada aba do `notebook`. Cada `Frame` conterá os widgets específicos de sua seção.
  - `self.notebook.add(self.client_frame, text="Clientes")`: Adiciona cada `Frame` como uma aba ao `notebook`, com o texto correspondente na aba.
  - `self.create_client_tab()`: Chama métodos separados para criar o conteúdo de cada aba. Isso ajuda a organizar o código e manter as funções menores e mais focadas.

Explicação Linha a Linha (Parte 2: Aba de Clientes)

- **`def create_client_tab(self):`** : Este método constrói a interface gráfica para a aba de gerenciamento de clientes.
    - **`Label(self.client_frame, text="Código:").grid(...)`** : Cria um rótulo (Label) com o texto "Código:" e o posiciona na grade (`grid`) dentro do `self.client_frame` . `grid` é um gerenciador de layout que organiza os widgets em uma grade de linhas e colunas. `padx` e `pady` adicionam preenchimento horizontal e vertical, e `sticky="w"` alinha o widget à esquerda (West).
    - **`self.client_codigo_entry = Entry(self.client_frame)`** : Cria um campo de entrada de texto (Entry) para o código do cliente.
    - **`self.client_codigo_entry.grid(...)`** : Posiciona o campo de entrada na grade. `sticky="ew"` faz com que o widget se expanda horizontalmente para preencher a célula.
    - **`self.client_codigo_entry.config(state=\'readonly\')`** : Configura o campo de código como somente leitura, pois o ID do cliente é gerado automaticamente pelo banco de dados.
    - **`Label(...)`** e **`Entry(...)`** : Padrão repetido para todos os campos de entrada de dados do cliente (Nome, CPF, Email, Telefone, Rua, CEP, Bairro, Cidade).
    - **`self.client_datanascimento_entry = DateEntry(...)`** : Cria um widget `DateEntry` para a data de nascimento, que oferece um seletor de calendário.
- **Botões para Cliente:**
    - **`Button(self.client_frame, text="Adicionar", command=self.add_client).grid(...)`** : Cria botões para as operações CRUD (Adicionar, Alterar, Deletar, Buscar) e um botão "Limpar". O argumento `command` vincula o botão a um método da classe `Application` (ex: `self.add_client`).
- **`Treeview for Client List`** : Cria uma tabela para exibir a lista de clientes.

- **`self.client_list = ttk.Treeview(self.client_frame, columns=(...), show="headings")`**: Cria um `Treeview` com as colunas especificadas. `show="headings"` significa que apenas os cabeçalhos das colunas serão visíveis, não a coluna padrão de hierarquia.
- **`self.client_list.grid(...)`**: Posiciona o `Treeview` na grade, ocupando várias colunas (`columnspan=5`) e se expandindo em todas as direções (`sticky="nsew"`).
- **`self.client_list.heading("id", text="ID")`**: Define o texto do cabeçalho para cada coluna.
- **`self.client_list.column("id", width=30)`**: Define a largura de cada coluna.
- **`self.client_list.bind("<Double-1>", self.on_double_click_client)`**: Associa um evento de duplo clique do botão esquerdo do mouse (`<Double-1>`) no `Treeview` ao método `self.on_double_click_client`. Isso permite que o usuário selecione um cliente na lista para edição.
- **`self.populate_client_list()`**: Chama um método para preencher a lista de clientes no `Treeview` ao iniciar a aba.

Métodos de Ação da Aba de Clientes:

- **`def add_client(self):`**: Lida com a adição de um novo cliente.
  - Obtém os dados dos campos de entrada.
  - Realiza validações usando `GUIComponents.validate_not_empty`, `validate_email`, `validate_cpf`, `validate_phone`, `validate_date`.
  - Se as validações passarem, cria um objeto `Client` com os dados.
  - Chama `self.client_manager.add_client(client)` para adicionar o cliente ao banco de dados.
  - Exibe uma mensagem de sucesso ou erro usando `GUIComponents.show_info` ou `show_error`.
  - Atualiza a lista de clientes (`self.populate_client_list()`) e limpa os campos (`self.clear_client_entries()`).

- `def update_client(self):`: Lida com a atualização de um cliente existente.
    - Obtém o ID do cliente do campo de código (que é somente leitura).
    - Obtém os dados atualizados dos outros campos de entrada.
    - Realiza validações.
    - Cria um objeto `Client` com o ID e os dados atualizados.
    - Chama `self.client_manager.update_client(client)`.
    - Exibe mensagem e atualiza a GUI.
- `def delete_client(self):`: Lida com a exclusão de um cliente.
    - Obtém o ID do cliente.
    - Pede confirmação ao usuário usando `GUIComponents.ask_yes_no`.
    - Se confirmado, chama `self.client_manager.delete_client(client_id)`.
    - Exibe mensagem e atualiza a GUI.
- `def search_client(self):`: Lida com a busca de clientes.
    - Obtém o termo de busca do campo de nome.
    - Chama `self.client_manager.search_client(name)`.
    - Limpa o `Treeview` e preenche-o com os resultados da busca.
- `def clear_client_entries(self):`: Limpa todos os campos de entrada da aba de clientes usando `GUIComponents.clear_entries`.
- `def populate_client_list(self):`: Preenche o `Treeview` com todos os clientes do banco de dados.
    - Limpa o conteúdo atual do `Treeview`.
    - Chama `self.client_manager.get_all_clients()` para obter a lista de clientes.
    - Para cada cliente, insere uma nova linha no `Treeview` com os dados do cliente.

- `def on_double_click_client(self, event):`: Evento acionado por um duplo clique no `Treeview` de clientes.
    - Obtém o item selecionado no `Treeview`.
    - Recupera os valores da linha selecionada.
    - Preenche os campos de entrada da aba de clientes com os dados do cliente selecionado, permitindo a edição.

Explicação Linha a Linha (Parte 3: Aba de Usuários)

- `def create_user_tab(self):`: Este método constrói a interface gráfica para a aba de gerenciamento de usuários.
    - Similar à aba de clientes, ele cria `Label`s e `Entry`s para os campos de usuário (Código, Nome, CPF, Email, Telefone, Data Nasc., Rua, CEP, Bairro, Cidade).
    - `self.user_senha_entry = Entry(self.user_frame, show="*")`: Cria um campo de entrada para a senha, onde `show="*"` faz com que os caracteres digitados sejam exibidos como asteriscos para ocultar a senha.
    - `self.user_tipo_combo = ttk.Combobox(self.user_frame, values=["admin", "vendedor", "financeiro", "estoque"])`: Cria um `Combobox` para o tipo de usuário, com valores pré-definidos para seleção.
    - `self.user_permissao_combo = ttk.Combobox(self.user_frame, values=["padrao", "avancado"])`: Cria um `Combobox` para o nível de permissão do usuário.
- **Botões para Usuário:**
    - Botões para as operações CRUD (Adicionar, Alterar, Deletar, Buscar) e Limpar, vinculados aos métodos correspondentes (`self.add_user`, `self.update_user`, etc.).
- `Treeview for User List:`
    - `self.user_list = ttk.Treeview(...)`: Cria um `Treeview` para exibir a lista de usuários, com colunas para todos os atributos do usuário, incluindo `tipo` e `permissao`.

- `self.user_list.bind("<Double-1>", self.on_double_click_user)`: Vincula o duplo clique ao método `self.on_double_click_user` para preencher os campos de entrada com os dados do usuário selecionado.
- `self.populate_user_list()`: Preenche a lista de usuários ao carregar a aba.

Métodos de Ação da Aba de Usuários:

- `def add_user(self):`: Adiciona um novo usuário.
  - Obtém os dados dos campos, incluindo a senha e os valores dos `Combobox`.
  - Realiza validações (similar aos clientes).
  - Cria um objeto `User`.
  - Chama `self.user_manager.add_user(user)`. A lógica de hashing da senha é tratada dentro do `UserManager`.
  - Exibe mensagem e atualiza a GUI.
- `def update_user(self):`: Atualiza um usuário existente.
  - Obtém o ID do usuário selecionado.
  - Obtém os dados atualizados. **Importante:** A senha é tratada de forma especial aqui. Se o campo de senha for preenchido, a nova senha será hashed pelo `UserManager`. Se for deixado vazio, a senha existente (hashed) será mantida.
  - Chama `self.user_manager.update_user(user)`.
  - Exibe mensagem e atualiza a GUI.
- `def delete_user(self):`: Deleta um usuário (com confirmação).
- `def search_user(self):`: Busca usuários pelo nome.
- `def clear_user_entries(self):`: Limpa os campos de entrada da aba de usuários.
- `def populate_user_list(self):`: Preenche o `Treeview` de usuários.

- `def on_double_click_user(self, event):`: Preenche os campos de entrada com os dados do usuário selecionado no `Treeview`.

Explicação Linha a Linha (Parte 4: Aba de Fornecedores)

- `def create_supplier_tab(self):`: Constrói a interface para a aba de gerenciamento de fornecedores.
    - Estrutura similar às abas anteriores, com `Label`s e `Entry`s para Código, Nome, CNPJ, Email, Telefone, Rua, CEP, Bairro, Cidade.
- **Botões para Fornecedor:**
    - Botões para operações CRUD e Limpar, vinculados a `self.add_supplier`, `self.update_supplier`, etc.
- `Treeview for Supplier List:`
    - `self.supplier_list = ttk.Treeview(...)`: Cria um `Treeview` para exibir a lista de fornecedores.
    - `self.supplier_list.bind("<Double-1>", self.on_double_click_supplier)`: Vincula o duplo clique ao método `self.on_double_click_supplier`.
    - `self.populate_supplier_list()`: Preenche a lista de fornecedores.

Métodos de Ação da Aba de Fornecedores:

- `def add_supplier(self):`: Adiciona um novo fornecedor.
- `def update_supplier(self):`: Atualiza um fornecedor existente.
- `def delete_supplier(self):`: Deleta um fornecedor.
- `def search_supplier(self):`: Busca fornecedores pelo nome.
- `def clear_supplier_entries(self):`: Limpa os campos de entrada da aba de fornecedores.
- `def populate_supplier_list(self):`: Preenche o `Treeview` de fornecedores.
- `def on_double_click_supplier(self, event):`: Preenche os campos de entrada com os dados do fornecedor selecionado.

Explicação Linha a Linha (Parte 5: Aba de Produtos)

- **`def create_product_tab(self):`**: Constrói a interface para a aba de gerenciamento de produtos.
    - `Label`s e `Entry`s para Código, Nome, Descrição, Preço, Quantidade.
    - **`self.product_fornecedor_combo = ttk.Combobox(self.product_frame, values=[])`**: Um `Combobox` para selecionar o fornecedor do produto. Inicialmente vazio, será preenchido dinamicamente.
- **Botões para Produto:**
    - Botões para operações CRUD e Limpar, vinculados a `self.add_product`, `self.update_product`, etc.
- **`Treeview for Product List`:**
    - **`self.product_list = ttk.Treeview(...)`**: Cria um `Treeview` para exibir a lista de produtos, incluindo colunas para `quantidade` e `fornecedor`.
    - **`self.product_list.bind("<Double-1>", self.on_double_click_product)`**: Vincula o duplo clique ao método `self.on_double_click_product`.
    - **`self.populate_product_list()`**: Preenche a lista de produtos.
    - **`self.populate_supplier_combobox()`**: Preenche o `Combobox` de fornecedores com os nomes dos fornecedores existentes no banco de dados.

Métodos de Ação da Aba de Produtos:

- **`def add_product(self):`**: Adiciona um novo produto.
    - Obtém os dados, incluindo o nome do fornecedor selecionado no `Combobox`.
    - **`supplier_id = next((s.id_fornecedor for s in self.supplier_manager.get_all_suppliers() if s.nome == supplier_name), None)`**: Busca o ID do fornecedor com base no nome

selecionado. Isso é importante porque o banco de dados armazena o ID do fornecedor, não o nome.
    - Cria um objeto `Product` e chama `self.product_manager.add_product(product, initial_stock)`.

- **`def update_product(self):`**: Atualiza um produto existente.
    - Similar ao `add_product`, obtém o ID do fornecedor.
    - Chama `self.product_manager.update_product(product)`.
- **`def delete_product(self):`**: Deleta um produto.
- **`def search_product(self):`**: Busca produtos pelo nome.
- **`def clear_product_entries(self):`**: Limpa os campos de entrada da aba de produtos.
- **`def populate_product_list(self):`**: Preenche o `Treeview` de produtos. Para cada produto, ele também busca o nome do fornecedor correspondente para exibição.
- **`def populate_supplier_combobox(self):`**: Preenche o `Combobox` de fornecedores com os nomes de todos os fornecedores cadastrados.
- **`def on_double_click_product(self, event):`**: Preenche os campos de entrada com os dados do produto selecionado.

Explicação Linha a Linha (Parte 6: Aba de Vendas)

- **`def create_sale_tab(self):`**: Constrói a interface para a aba de vendas.
    - `Label`s e `Entry`s para Código da Venda, Quantidade, Preço Unitário, Total.
    - **`self.sale_cliente_combo = ttk.Combobox(...)`**: `Combobox` para selecionar o cliente da venda.
    - **`self.sale_produto_combo = ttk.Combobox(...)`**: `Combobox` para selecionar o produto a ser vendido.
    - Os campos de Preço Unitário e Total são `readonly` e serão preenchidos automaticamente.

- **Botões para Venda:**
  - `Button(..., text="Adicionar Item", command=self.add_sale_item)`: Adiciona um item à lista de itens da venda.
  - `Button(..., text="Finalizar Venda", command=self.finalize_sale)`: Finaliza a venda, registrando-a no banco de dados.
  - `Button(..., text="Limpar Venda", command=self.clear_sale_entries)`: Limpa os campos e a lista de itens da venda atual.
- `Treeview for Sale Items:`
  - `self.sale_items_list = ttk.Treeview(...)`: Cria um `Treeview` para exibir os itens que estão sendo adicionados à venda atual (Produto, Quantidade, Preço Unitário, Total Item).
- `self.populate_client_combobox()` e `self.populate_product_combobox()`: Preenchem os `Combobox` de clientes e produtos ao carregar a aba.

Métodos de Ação da Aba de Vendas:

- `def add_sale_item(self):`: Adiciona um produto à lista de itens da venda.
  - Obtém o produto e a quantidade selecionados.
  - Calcula o preço unitário e o total do item.
  - Adiciona o item ao `self.sale_items_list`.
  - Atualiza o total geral da venda.
- `def finalize_sale(self):`: Finaliza a venda.
  - Obtém o cliente selecionado e o usuário logado (assumindo que o usuário logado é o que está usando a aplicação).
  - Percorre os itens no `self.sale_items_list`.
  - Para cada item, registra a venda no banco de dados (seria necessário um `SaleManager` e `SaleItemManager` para uma implementação completa).
  - Atualiza o estoque do produto vendido (`self.product_manager.update_stock`).

    - Exibe mensagem de sucesso e limpa a venda.

- **`def clear_sale_entries(self):`**: Limpa os campos e o `Treeview` da aba de vendas.
- **`def populate_client_combobox(self):`**: Preenche o `Combobox` de clientes.
- **`def populate_product_combobox(self):`**: Preenche o `Combobox` de produtos.
- **`def on_product_select(self, event):`**: (Não presente no trecho, mas esperado) Um método que seria vinculado ao evento de seleção do `self.sale_produto_combo` para preencher automaticamente o `Preço Unitário` do produto selecionado.

Considerações Finais sobre **`main_app.py`**:

O `main_app.py` é o coração da interface do usuário. Ele demonstra como os diferentes componentes (widgets Tkinter, classes de gerenciamento de negócios e modelos de dados) são orquestrados para criar uma aplicação funcional. A modularização em métodos separados para cada aba (`create_client_tab`, etc.) e para cada ação (`add_client`, etc.) torna o código mais organizado e fácil de entender e manter. A integração com `GUIComponents` para validações e mensagens padroniza a interação com o usuário. Para uma aplicação de vendas completa, seria necessário implementar as classes de gerenciamento para Vendas, Compras e Financeiro, e expandir a lógica de interação na aba de vendas para registrar os itens da venda individualmente e atualizar o estoque de forma mais robusta.