

Arquivo: model.py

```
from modules.atas.database import DatabaseATASManager
from PyQt6.QtWidgets import *
from PyQt6.QtGui import *
from PyQt6.QtCore import *
import os
import pandas as pd
from PyQt6.QtSql import QSqlDatabase, QSqlTableModel, QSqlQuery
import logging
import sqlite3
import sys
import subprocess

class GerarAtasModel(QObject):
    tabelaCarregada = pyqtSignal()

    def __init__(self, database_path, parent=None):
        super().__init__(parent)
        self.database_ata_manager = DatabaseATASManager(database_path)
        self.db = None # Adiciona um atributo para o banco de dados
        self.model = None # Atributo para o modelo SQL
        self.init_database() # Inicializa a conexão e a estrutura do banco de dados

    def init_database(self):
        if QSqlDatabase.contains("my_conn"):
            QSqlDatabase.removeDatabase("my_conn")

        db_path = str(self.database_ata_manager.db_path)
        self.db = QSqlDatabase.addDatabase('QSQLITE', "my_conn")
        self.db.setDatabaseName(db_path)

        if not self.db.open():
            print("Não foi possível abrir a conexão com o banco de dados.")
        else:
            print("Conexão com o banco de dados aberta com sucesso.")
            self.adjust_table_atas_structure()

    def setup_model(self, table_name, editable=False):
        """Configura o modelo SQL para a tabela especificada."""
        self.model = CustomSqlTableModel(
            parent=self, db=self.db, database_manager=self.database_ata_manager,
            non_editable_columns=[4, 8, 10, 13], gerar_atas_model=self
        )
        self.model.setTable(table_name)
        self.model.setEditStrategy(QSqlTableModel.EditStrategy.OnFieldChange)
        self.model.select()
        return self.model

    def create_table_if_not_exists(self):
        """Cria a tabela 'controle_dispensas' com a estrutura definida, caso ainda não exista."""

```

```

query = QSqlQuery(self.db)
if not query.exec("""
    CREATE TABLE IF NOT EXISTS controle_atas (
        grupo TEXT,
        item TEXT PRIMARY KEY,
        catalogo TEXT,
        descricao TEXT,
        descricao_detalhada TEXT,
        unidade TEXT,
        quantidade TEXT,
        valor_estimado TEXT,
        valor_homologado_item_unitario TEXT,
        percentual_desconto TEXT,
        valor_estimado_total_do_item TEXT,
        valor_homologado_total_item TEXT,
        marca_fabricante TEXT,
        modelo_versao TEXT,
        situacao TEXT,
        uasg TEXT,
        orgao_responsavel TEXT,
        num_pregao TEXT,ano_pregao TEXT,
        srp TEXT,
        objeto TEXT,
        melhor_lance TEXT,
        valor_negociado TEXT,
        ordenador_despesa TEXT,
        empresa TEXT,
        cnpj TEXT
    )
"""):

    print("Falha ao criar a tabela 'controle_atas':", query.lastError().text())
else:
    print("Tabela 'controle_atas' criada com sucesso.")

def adjust_table_atas_structure(self):
    query = QSqlQuery(self.db)
    if not query.exec("SELECT name FROM sqlite_master WHERE type='table' AND name='controle_atas'"):

        print("Erro ao verificar existência da tabela:", query.lastError().text())
    if not query.next():

        print("Tabela 'controle_atas' não existe. Criando tabela...")
        self.create_table_if_not_exists()
    else:
        print("Tabela 'controle_atas' existe. Verificando estrutura da coluna...")

def configure_columns(self, table_view, visible_columns):
    for column in range(self.model.columnCount()):
        header = self.model.headerData(column, Qt.Orientation.Horizontal)
        if column not in visible_columns:
            table_view.hideColumn(column)
        else:
            self.model.setHeaderData(column, Qt.Orientation.Horizontal, header)

```

```

class CustomSqlTableModel(QSqlTableModel):
    tabelaCarregada = pyqtSignal()

    def __init__(self, parent=None, db=None, database_manager=None, non_editable_columns=None, gerar_atas_model=None):
        super().__init__(parent, db)
        self.database_manager = database_manager
        self.non_editable_columns = non_editable_columns if non_editable_columns is not None else []
        self.gerar_atas_model = gerar_atas_model
        self.connection = None

        # Define os nomes das colunas
        self.column_names = [
            'grupo', 'item', 'catalogo', 'descricao', 'unidade', 'quantidade', 'valor_estimado',
            'valor_homologado_item_unitario', 'percentual_desconto',
            'valor_estimado_total_do_item',
            'valor_homologado_total_item', 'marca_fabricante', 'modelo_versao', 'situacao',
            'descricao_detalhada', 'uasg', 'orgao_responsavel', 'num_pregao', 'ano_pregao',
            'srp', 'objeto', 'melhor_lance', 'valor_negociado', 'ordenador_despesa', 'empresa',
            'cnpj'
        ]
    }

    def flags(self, index):
        if index.column() in self.non_editable_columns:
            return super().flags(index) & ~Qt.ItemFlag.ItemIsEditable  # Remove a permissão de edição
        return super().flags(index)

    def data(self, index, role=Qt.ItemDataRole.DisplayRole):
        # Verifica se a coluna deve ser não editável e ajusta o retorno para DisplayRole
        if role == Qt.ItemDataRole.DisplayRole and index.column() in self.non_editable_columns:
            return super().data(index, role)

        return super().data(index, role)

    def connect_to_database(self):
        if self.connection is None:
            self.connection = sqlite3.connect(self.database_manager.db_path)  # Corriga para o atributo correto
            logging.info(f"Conexão com o banco de dados aberta em {self.database_manager.db_path}")
        return self.connection

    def close_connection(self):
        if self.connection:
            logging.info("Fechando conexão com o banco de dados...")
            self.connection.close()
            self.connection = None
            logging.info(f"Conexão com o banco de dados fechada em {self.database_manager}")

```

```

def execute_query(self, query, params=None):
    conn = self.connect_to_database()
    try:
        cursor = conn.cursor()
        if params:
            cursor.execute(query, params)
        else:
            cursor.execute(query)
        conn.commit()
        return cursor.fetchall()
    except sqlite3.Error as e:
        logging.error(f"Erro ao executar consulta: {query}, Erro: {e}")
        return None
    finally:
        self.close_connection()

def carregar_tabela(self):
    try:
        # 1. Seleciona o arquivo (esta parte está igual, funciona perfeitamente)
        caminho_arquivo, _ = QFileDialog.getOpenFileName(
            None,
            "Carregar Tabela",
            "",
            "Arquivos Excel (*.xlsx);;Todos os Arquivos (*)"
        )
        if not caminho_arquivo:
            return # Usuário cancelou, então não fazemos nada

        # 2. Carrega o arquivo Excel
        tabela = pd.read_excel(caminho_arquivo)

        # 3. Garante que todas as colunas do banco de dados existam no DataFrame.
        #     Se uma coluna não existir na planilha, ela é criada com valores vazios.
        for col in self.column_names: # self.column_names já existe na sua classe
            if col not in tabela.columns:
                tabela[col] = None

        # Garante que as colunas no DataFrame estejam na mesma ordem que no banco
        tabela = tabela[self.column_names]

        # 4. Chama a nova função centralizada para fazer o trabalho no banco de dados
        #     O self.database_manager é o objeto que contém a nova função.
        sucesso, mensagem = self.database_manager.substituir_dados_controle_atas(tabela)

        # 5. Informa o usuário e atualiza a tela
        if sucesso:
            QMessageBox.information(None, "Sucesso", mensagem)
            # Atualiza o modelo na tela para refletir as mudanças do banco
            self.select()
            self.tabelaCarregada.emit()
        else:
            QMessageBox.critical(None, "Erro de Banco de Dados", mensagem)
    except Exception as e:
        logging.error(f"Erro ao carregar a tabela: {e}")

```

```
except FileNotFoundError:
    QMessageBox.critical(None, "Erro", "Arquivo não encontrado. Verifique o caminho.")
except Exception as e:
    QMessageBox.critical(None, "Erro Crítico", f"Ocorreu um erro inesperado ao carregar a
planilha: {e}")

def abrir_tabela_nova(self):
    # Define o caminho do arquivo Excel
    file_path = os.path.join(os.getcwd(), "tabela_nova.xlsx")

    # Cria um DataFrame vazio com as colunas especificadas
    df = pd.DataFrame(columns=["item", "catalogo", "descricao", "descricao_detalhada"])

    try:
        # Tenta salvar o DataFrame no arquivo Excel
        df.to_excel(file_path, index=False)

        # Abre o arquivo Excel após a criação
        if sys.platform == "win32":
            os.startfile(file_path)
        elif sys.platform == "darwin": # macOS
            subprocess.Popen(["open", file_path])
        else: # Linux
            subprocess.Popen(["xdg-open", file_path])

    except PermissionError:
        # Mostra uma mensagem para o usuário caso o arquivo já esteja aberto
        QMessageBox.warning(None, "Aviso", "O arquivo 'tabela_nova.xlsx' já está aberto. Por
favor, feche-o e tente novamente.")
```