Estudo Dirigido 005

Banco de Dados e Programação Orientada a Objetos

Aplicação: Sistema de Controle de Garantia de Equipamentos

Prazo de entrega: 24/10
Entrega: Repositório individual no GitHub (público)
Banco: PostgreSQL (via DBeaver)
Linguagem: Python 3.x

---

1. Contexto e Propósito

Este estudo dirigido tem como objetivo conduzir o aluno na construção de um módulo de **gestão de garantias de equipamentos**, parte de um projeto que visa o desenvolvimento de uma aplicação completa, com camadas de dados e lógica de negócio.

O problema proposto é recorrente: usuários perdem notas fiscais e certificados de garantia.
A aplicação deverá permitir o **cadastro de equipamentos**, **armazenamento dos documentos de compra** e **aviso sobre o vencimento da garantia**.

O foco é o domínio da **camada de dados** (banco de dados) e da **camada de lógica** (POO), preparando terreno para integração futura com o framework web.

---

2. Objetivos de Aprendizagem

Ao concluir este estudo, você deverá ser capaz de:

1. Modelar e implementar um banco de dados relacional em PostgreSQL.
2. Compreender entidades, atributos e chaves primárias/estrangeiras.
3. Manipular dados via SQL com consultas simples e compostas.
4. Representar entidades do banco em classes Python (POO).
5. Relacionar a camada de persistência com a camada de negócio.
6. Versionar e documentar corretamente o código no GitHub.

---

3. Estrutura Esperada no GitHub

Cada aluno deverá criar um repositório com o nome:

`ed005_garantia_nomeAluno`

Estrutura recomendada:

```
ed005_garantia_nomeAluno/
│
├── sql/
│   ├── schema.sql
│   ├── inserts.sql
│
├── src/
│   ├── main.py
│   ├── database.py
│   ├── models/
│   │   ├── equipamento.py
│   │   ├── garantia.py
│   │   └── loja.py
│
├── prints/
│   ├── modelo_logico.png
│   ├── consultas_dbeaver.png
│   ├── execucao_terminal.png
│
└── README.md
```

---

4. Etapas de Desenvolvimento

4.1 Modelo Lógico – Concepção e Justificativa

1. Identifique e descreva as **entidades principais** da aplicação: *Loja*, *Equipamento* e *Garantia*.
2. Liste os **atributos** de cada entidade e justifique sua escolha.
3. Determine as **chaves primárias e estrangeiras**.
4. Crie o **diagrama lógico** (Draw.io, Miro ou similar) e salve como `prints/modelo_logico.png`.
5. Descreva no README uma **justificativa técnica** para as relações criadas (ex: "Uma loja pode vender vários equipamentos, logo a relação é 1:N").

Reflexão: explique a diferença entre modelo conceitual, lógico e físico.

4.2 Modelo Físico – Implementação no PostgreSQL

1. Crie o banco `app_garantia` no DBeaver.
2. Construa o script `schema.sql` contendo as tabelas:
    - `loja`
    - `equipamento`
    - `garantia`

Inclua restrições como `NOT NULL`, `CHECK`, `UNIQUE`, `ON DELETE RESTRICT`.

3. Insira ao menos três registros em cada tabela (`inserts.sql`).
4. Capture o resultado no DBeaver (`prints/consultas_dbeaver.png`).

Atividade investigativa: pesquise e aplique o uso de `ON DELETE CASCADE` e explique quando ele é útil ou perigoso.

4.3 Consultas SQL – Análise de Dados

Crie e documente no README consultas que respondam:

1. Quais equipamentos estão vinculados a cada loja?
2. Quais garantias vencem nos próximos 30 dias?
3. Qual loja possui o maior número de garantias vencidas?
4. Qual o tempo médio de garantia por loja?

Comente como cada consulta poderia ser usada no contexto da aplicação (por exemplo, para relatórios ou alertas).

4.4 POO – Representação das Entidades em Python

Crie classes correspondentes às tabelas dentro da pasta `models/`.

Cada classe deve conter:

- Construtor (`__init__`)
- Método `__str__` para exibição legível
- Comentários de documentação

Exemplo:

```
class Equipamento:
    def __init__(self, id, nome, data_compra, preco):
        self.id = id
        self.nome = nome
        self.data_compra = data_compra
        self.preco = preco

    def __str__(self):
        return f"{self.nome} ({self.data_compra})"
```

Desafio opcional: implemente uma classe `GarantiaEstendida` herdando de `Garantia`.

4.5 Camada de Persistência – Conexão e CRUD

Crie o módulo `database.py` com uma classe `Database` responsável por:

- Estabelecer conexão com PostgreSQL (`psycopg2`);
- Executar consultas SQL e retornar resultados;
- Tratar erros com `try/except`.

Exemplo base:

```
import psycopg2

class Database:
    def __init__(self):
        self.conn = psycopg2.connect(
            dbname="app_garantia",
            user="postgres",
            password="sua_senha",
            host="localhost",
            port="5432"
        )

    def consultar(self, query):
        try:
            cur = self.conn.cursor()
            cur.execute(query)
            return cur.fetchall()
        except Exception as e:
            print("Erro na consulta:", e)
```

```
        finally:
            cur.close()
```

Implemente no `main.py`:

- Uma consulta que retorne equipamentos e garantias;
- Impressão dos resultados em formato legível.

4.6 Testes e Documentação

1. Capture a execução no terminal (`prints/execucao_terminal.png`).
2. Documente no README:
    - Estrutura do banco e relações;
    - Prints das consultas;
    - Explicação do funcionamento do código;
    - Reflexão pessoal:
        - O que aprendi neste estudo?
        - Que erros enfrentei e como resolvi?
        - Como este exercício se conecta ao projeto integrador?

---

5. Incrementos Interdisciplinares

1. **Padrões de Projeto (MVC):**
    - Banco de dados → Model
    - Classes Python → Controller
    - Futuras views (APIs ou HTML) → View
2. **Metodologias Ágeis:**
    - Descreva como dividiu seu trabalho em pequenas tarefas (sprints).
3. **Testes de Software:**
    - Crie um teste simples com `assert` verificando se a consulta retorna resultados válidos.

---

6. Rubricas de Avaliação (Total: 10 pontos)

Critério	Descrição	Peso

Critério	Descrição	Peso
Modelagem Lógica e Justificativa	Entidades, relacionamentos e coerência com o problema	2
Implementação SQL e Integridade	Scripts funcionais, uso de restrições, chaves e relacionamentos	2
Consultas SQL Analíticas	Correção e interpretação das consultas	2
POO e Camada de Persistência	Classes, encapsulamento, conexão e consultas	2
Documentação e GitHub	Organização do repositório, README completo e prints	2

Bônus (+1 ponto):

- CRUD completo (Create, Read, Update, Delete);
- Cálculo automático de vencimento da garantia no Python;
- Uso de variáveis de ambiente (.env) para as credenciais.

---

7. Boas Práticas de Versionamento

1. Realize commits pequenos e descritivos:
    - `git commit -m "criação do schema.sql"`
    - `git commit -m "implementação da classe Equipamento"`
2. Inclua um `.gitignore` com:

```
__pycache__/
*.pyc
.env
```

3. Utilize o README.md como documento central de explicação do projeto.

---

8. Entrega Final

- Enviar o link do repositório até **24/10**.
- O projeto deve estar funcional, documentado e versionado.
- A avaliação considerará a funcionalidade, clareza técnica e reflexão individual.