Projeto Prático: Sistema de Gerenciamento de Eventos

Objetivo: Integrar e aplicar os conceitos do curso em um projeto de back-end completo.

Tema: Sistema de Gerenciamento de Eventos.

Formato: Desenvolvimento em equipe (3-4 pessoas), com gerenciamento de código via GitHub.

Descrição do Projeto

O sistema deve simular o controle de eventos como workshops, palestras e meetups. Ele permitirá o cadastro de eventos, a inscrição de participantes e a emissão de relatórios detalhados, funcionando como a "espinha dorsal" de uma plataforma de gestão.

Requisitos Funcionais

- **Cadastro de Eventos:**
  - **Atributos:** nome, data, local, capacidade máxima, categoria (ex: Tech, Marketing), preço do ingresso.
  - **Validação:** A data do evento não pode ser anterior à data atual. A capacidade máxima deve ser um número positivo.
- **Inscrição de Participantes:**
  - **Atributos:** nome, email, evento inscrito.
  - **Regras:**
    - Não permitir inscrições se o evento estiver lotado.
    - Um mesmo e-mail não pode ser cadastrado mais de uma vez no mesmo evento.
- **Gerenciamento do Sistema:**
  - Listar todos os eventos e suas informações.
  - Buscar eventos por categoria ou por data.
  - Cancelar a inscrição de um participante.
  - Funcionalidade de `check-in` para marcar a presença do participante no evento.
- **Relatórios e Análises:**
  - Número total de inscritos por evento.
  - Lista de eventos com vagas disponíveis.
  - Calcular a receita total de um evento específico.

Requisitos Não Funcionais

- **Programação Orientada a Objetos (POO):** O projeto deve ser estruturado em classes.
  - **Classes:** `Evento` (superclasse), `Workshop` e `Palestra` (subclasses), `Participante` e `SistemaEventos` (a classe principal que gerencia as operações).

    - **Herança:** Use herança para que `Workshop` e `Palestra` herdem de `Evento` e adicionem atributos específicos (ex: `Workshop` pode ter `material_necessário`).
    - **Polimorfismo:** Crie um método, como `detalhes()`, que se comporte de maneira diferente em `Workshop` e `Palestra`.
    - **Encapsulamento:** Mantenha os atributos de classe como privados (`__atributo`) e utilize `getters` e `setters` para acesso controlado.
- **Persistência de Dados:** Utilize um banco de dados simples, como **SQLite3**, ou armazene os dados em um arquivo `.json` para que as informações não se percam ao desligar o programa.
- **Testes:** Crie no mínimo 10 testes unitários, usando a biblioteca `unittest`, para garantir o funcionamento correto de funcionalidades críticas (ex: validação de data, controle de vagas, `check-in`, cancelamento).
- **Controle de Versão:** Utilize o Git com um histórico de commits claro. O desenvolvimento deve ser feito em *branches* separadas, com a integração no branch principal (`main`) via *pull requests*.

Documentação

- **`README.md`:** Incluir uma descrição do projeto, instruções de instalação de dependências e como rodar o sistema.
- **Diagrama de Classes UML:** Desenhe o diagrama para mostrar a estrutura e a relação entre as classes.
- **Casos de Uso:** Descrever cenários de interação do usuário com o sistema (ex: "Como um usuário, eu quero me inscrever em um evento para garantir minha vaga").

Fases do Projeto

- **Fase 1: Planejamento:**
    - Definir os papéis na equipe (desenvolvedores, QA, etc.).
    - Criar o *Product Backlog* com todas as funcionalidades.
    - Entregar o diagrama de classes UML e os casos de uso.
- **Fase 2: Desenvolvimento:**
    - Implementar as classes e funcionalidades, incluindo a persistência de dados.
    - Utilizar o Git para versionamento, com *branches* para cada funcionalidade e *pull requests* para a integração.
- **Fase 3: Testes e Integração:**
    - Escrever os testes unitários.
    - Integrar todas as funcionalidades no branch `main`.
- **Fase 4: Documentação e Entrega:**
    - Finalizar o `README.md`.
    - Preparar um relatório final com o que foi aprendido, dificuldades e melhorias.
    - Entregar o link do repositório GitHub e a documentação em PDF.