FGV EMAp
ESCOLA DE MATEMÁTICA APLICADA

Prova 4 de Engenharia de Software
28 de Outubro de 2025

Nome: _____

Instruções Gerais

- Todas as questões da prova devem ser organizadas em **dois arquivos apenas**:
  1. Um módulo denominado `funcoes.py`, no qual deverão ser implementadas todas as funções desenvolvidas ao longo da prova;
  2. Um arquivo `main.py`, que atuará como **driver code**, responsável por importar o módulo `funcoes.py` e demonstrar o funcionamento das funções nele implementadas.
- O código deverá apresentar-se **claro, devidamente organizado e adequadamente documentado**. Apenas serão aceitas soluções que evidenciem legibilidade, coerência estrutural e rigor metodológico.

Questão 1 (5 pontos) Você precisa garantir uma única instância de um componente de configuração (por exemplo, ConfigService) durante toda a execução do programa.

Três alternativas comuns de Singleton em Python:

1. Módulo como Singleton. Em Python, cada módulo é carregado uma única vez. Colocar o estado único diretamente em variáveis do módulo já produz, na prática, um singleton simples, sem classe. É a opção mais direta e enxuta, mas tende a espalhar dependências globais e pode dificultar o reset em testes (exige recarregar o módulo ou sobrescrever variáveis manualmente).
2. Classe com atributo de classe e inicialização sob demanda (lazy initialization). Cria-se uma classe que guarda a instância única em um atributo de classe (por exemplo, `_inst`). Um método de classe `get_instance()` retorna a instância; se ainda não existir, cria no momento da primeira chamada (lazy). Fica explícito onde a instância nasce e é simples introduzir um ponto de reset para testes (por exemplo, um método de classe que zera `_inst`).
3. Classe com `\new` customizado. A lógica de manter uma única instância é colocada no construtor de baixo nível `__new__`, chamado antes de `__init__`. Se a instância já existir, `__new__` apenas a retorna; caso contrário, cria e guarda. A vantagem é o encapsulamento no ciclo de criação do objeto; a desvantagem é ser menos explícito para quem lê o código e, em testes, pode exigir um método auxiliar de reset.

Tarefas.

(a) Conceitual – 2 pontos. Explique, em 6–10 linhas, vantagens e desvantagens das três alternativas acima, considerando: simplicidade e clareza de uso; testabilidade (como reinicializar o estado único); momento de criação (no import do módulo versus lazy na primeira chamada); acoplamento (uso de globais e dependências implícitas).

(b) Implementação mínima – 3 pontos. Implemente uma alternativa (2 ou 3) chamada `ConfigServiceSingleton`, expondo `get_instance()` que retorna a única instância, criando-a apenas na primeira chamada, e um mecanismo simples de reset para testes (por exemplo, um método de classe `_reset_for_tests()` que zera a instância). No `main.py`, demonstre o uso chamando `get_instance()` duas vezes e exibindo que o `id()` do objeto é o mesmo nas duas chamadas.

Questão 2 (5 pontos)

Sua plataforma precisa criar componentes de infraestrutura alinhados ao provedor de banco de dados escolhido (ex.: SQLite, Postgres). Além do Repository (que fala com o banco), há produtos relacionados que também dependem do mesmo provedor para manter compatibilidade de família:

- Repository – encapsula acesso a dados (conexão/queries).
- TransactionManager – inicia/fecha transações de forma específica do banco.
- MigrationTool – aplica migrações no dialeto do banco.
- CacheClient (opcional) – compatibilidade com o dialeto e convenções do provedor.

O que são "produtos relacionados"? São componentes de uma mesma família tecnológica que precisam ser coerentes entre si (ex.: RepositoryPostgres, TransactionManagerPostgres, MigrationToolPostgres). Eles compartilham suposições sobre dialeto SQL, semântica de transações, limites de tipos, etc.

O que significa "compatibilidade de família"? Quando o provedor de banco muda (SQLite → Postgres), todos os produtos dessa família devem mudar juntos para a variante correspondente. Não pode existir uma combinação cruzada incoerente (ex.: RepositoryPostgres com TransactionManagerSQLite).

Tarefas.

(a) Conceitual – 2 pts. Em 6–10 linhas, explique quando Factory Method é suficiente e quando Abstract Factory é mais adequado, neste cenário. Inclua:

- O que muda quando surgem múltiplos produtos correlatos (por exemplo, `Repository`, `CacheClient`) que variam em conjunto.
- Como cada padrão afeta extensibilidade e acoplamento.

(b) Esboço de desenho – 3 pts. No `funcoes.py`, escreva esqueletos (interfaces/ABCs e classes mínimas) para:

- Uma hierarquia com Factory Method para criar `Repository` conforme o banco (uma fábrica por *Creator*); *ou*
- Uma Abstract Factory com família de produtos `Repository` e `CacheClient` (ambos variam por banco).

Você pode utilizar `abstractmethod` ou *stubs* (`pass`/`raise NotImplementedError`).