Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital
Disciplina: Boas Práticas de Programação
Semestre: 2024.2 - **Turma:** 6M56
Professor: Edson Jackson de Medeiros Neto

Estudo Dirigido 04 - Testes e Depuração

Objetivo

Este trabalho tem como objetivo:

- A aplicação prática de testes unitários e de integração.
- O uso de ferramentas de depuração, incluindo *loggers* e *breakpoints*.
- Desenvolver uma solução funcional em uma linguagem de sua escolha: **Python**, **Java** ou **C++**.

Descrição do Problema

Deve-se implementar um sistema para gerenciar o processamento de pedidos de uma loja virtual. O sistema deve contemplar as seguintes funcionalidades:

Catálogo de Produtos

- Cada produto possui:
  - **ID**: um identificador único (inteiro).
  - **Nome**: o nome do produto (texto).
  - **Preço**: o preço do produto (número decimal).
- Deve ser possível:
  - Adicionar produtos ao catálogo.
  - Listar todos os produtos disponíveis.
  - Buscar um produto pelo seu **ID**.

Pedido

- Cada pedido é composto por:
  - Uma lista de produtos escolhidos.
  - O nome do cliente que realizou o pedido.
- O sistema deve calcular o valor total do pedido.
- Se o valor total do pedido ultrapassar **R$ 100,00**, um desconto de **10%** deve ser aplicado.

Simulação de Banco de Dados

Para simplificar, os dados do catálogo e dos pedidos devem ser armazenados em memória por meio de estruturas de classes ou coleções, simulando o comportamento de um banco de dados.

Detalhes de Implementação

Implemente as seguintes classes e métodos:

Classe Produto

- Propriedades:
  - **id**: identificador único (`int`).
  - **nome**: nome do produto (`String` ou equivalente).
  - **preco**: preço do produto (`float` ou equivalente).
- Métodos:
  - Construtor para inicializar os valores do produto.

Classe Catálogo

- Métodos:
  - `adicionarProduto(Produto produto)`: adiciona um produto ao catálogo.
  - `listarProdutos()`: retorna a lista de produtos.
  - `buscarProdutoPorId(int id)`: retorna o produto com base no **ID**.

Classe Pedido

- Propriedades:
  - **produtos**: lista de produtos adicionados ao pedido.
  - **cliente**: nome do cliente.
- Métodos:
  - `adicionarProduto(Produto produto)`: adiciona um produto ao pedido.
  - `calcularTotal()`: calcula o valor total do pedido, aplicando o desconto se necessário.

Testes

Testes Unitários

- Verifique a adição e busca de produtos no catálogo.
- Teste o cálculo do valor total do pedido, com e sem aplicação de desconto.

Testes de Integração

- Simule um cenário completo, com:
  - Adição de produtos ao catálogo.
  - Criação de um pedido com múltiplos produtos.
  - Cálculo do valor total do pedido.

Testes e Ferramentas

Bibliotecas para Testes Unitários e Mocking

Para a implementação dos testes unitários e de integração, bem como para a simulação de objetos com *mocking*, utilize as bibliotecas descritas abaixo, de acordo com a linguagem escolhida:

Python

- **Teste Unitário**:
  - Utilize a biblioteca integrada `unittest` ou a biblioteca moderna `pytest`.
  - `unittest`: Faz parte da biblioteca padrão do Python e oferece suporte básico para testes unitários.
  - `pytest`: Uma biblioteca poderosa e extensível, com suporte a *fixtures*, parametrização e um ecossistema de plugins.
- **Mocking**:
  - Utilize o módulo `unittest.mock`, já integrado ao Python, ou o plugin `pytest-mock` para integração com `pytest`.

Java

- **Teste Unitário**:
  - Utilize o **JUnit 5 (Jupiter)**, uma biblioteca moderna e amplamente utilizada para testes unitários em Java.
  - Configuração: Inclua a dependência no `pom.xml` do Maven ou no arquivo de configuração do Gradle.
- **Mocking**:
  - Utilize a biblioteca `Mockito` para criar objetos simulados e verificar interações.

C++

- **Teste Unitário**:
  - Utilize o **Google Test**, uma biblioteca robusta para criação de testes unitários.
  - Configuração: Faça o *clone* do repositório oficial e configure com o `CMake`.
- **Mocking**:
  - Utilize o **Google Mock**, que é integrado ao Google Test, para simulação de objetos e verificação de interações.

Como Configurar

Python

- Instale as bibliotecas necessárias com o comando:

```
pip install pytest pytest-mock
```

- Caso opte pelo `unittest`, não é necessária instalação adicional, pois já faz parte da biblioteca padrão do Python.

Java

- Adicione as dependências no arquivo `pom.xml` do Maven:

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.9.3</version>
    <scope>test</scope>
</dependency>
<dependency>
```

```
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>4.11.0</version>
        <scope>test</scope>
    </dependency>
```

C++

- Baixe e configure o **Google Test** e o **Google Mock**:

```
git clone https://github.com/google/googletest.git
cmake -S . -B build
cmake --build build
```

- Inclua a biblioteca como parte do ambiente de compilação do seu projeto.

Resumo Completo das Ferramentas

A tabela abaixo resume as bibliotecas a serem utilizadas para cada linguagem:

Linguagem	Biblioteca de Teste Unitário	Biblioteca de Mocking
Python	`unittest` ou `pytest`	`unittest.mock` ou `pytest-mock`
Java	`JUnit 5`	`Mockito`
C++	`Google Test`	`Google Mock`

Tabela 1: Ferramentas recomendadas para testes e *mocking*.

Depuração

Logger

Adicione *loggers* para registrar as seguintes ações:

- Adição de produtos ao catálogo.
- Criação de pedidos.
- Aplicação de descontos.

Debug com Breakpoints

Configure *breakpoints* em sua IDE para depurar:

- A aplicação de descontos.
- A adição de produtos ao pedido.

Depuração e Loggers

Para auxiliar na depuração do código e no rastreamento de informações durante a execução, utilize *loggers* para registrar eventos importantes, como:

- Adição de produtos ao catálogo.
- Criação de pedidos.
- Aplicação de descontos.
- Mensagens de erro ou comportamento inesperado.

Abaixo estão descritas as ferramentas de *logging* recomendadas para cada linguagem, com exemplos de configuração e uso.

Python

- Utilize a biblioteca padrão `logging`, que fornece uma API poderosa para gerenciamento de logs.
- Configuração básica do logger:

```
import logging

# Configura  o do logger
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')

# Uso do logger
logging.info("Produto adicionado ao cat logo")
logging.error("Erro ao buscar produto por ID")
```

Java

- Utilize a biblioteca `java.util.logging`, integrada ao JDK, ou frameworks avançados como `Log4j`.
- Exemplo básico com `java.util.logging`:

```
import java.util.logging.Logger;

public class App {
    private static final Logger logger = Logger.getLogger(App.class.getName());

    public static void main(String[] args) {
        logger.info("Produto adicionado ao cat logo");
        logger.warning("Desconto n  o aplicado devido a erro");
    }
}
```

- Para projetos maiores, recomenda-se o uso do `Log4j` ou `SLF4J` devido às suas funcionalidades avançadas.

C++

- Utilize bibliotecas populares como `spdlog` ou mensagens simples com `std::cout` para projetos pequenos.
- Exemplo com `spdlog`:

```
#include <spdlog/spdlog.h>

int main() {
    spdlog::info("Produto adicionado ao cat logo");
    spdlog::error("Erro ao buscar produto por ID");
    return 0;
}
```

- Para projetos simples, é possível utilizar diretamente:

```
#include <iostream>

int main() {
    std::cout << "Produto adicionado ao cat logo" << std::endl;
    std::cerr << "Erro ao buscar produto por ID" << std::endl;
    return 0;
}
```

Resumo Completo das Ferramentas para Loggers

A tabela abaixo resume as bibliotecas recomendadas para *logging* em cada linguagem:

Linguagem	**Biblioteca de Logger**
Python	`logging`
Java	`java.util.logging`, `Log4j`, `SLF4J`
C++	`spdlog`, `std::cout/std::cerr`

Tabela 2: Ferramentas recomendadas para *logging*.

Como Configurar os Loggers

Python

- A biblioteca `logging` já está integrada ao Python, não sendo necessária instalação adicional.
- Para configurações mais avançadas, como envio de logs para arquivos, consulte a documentação oficial.

Java

- Para `java.util.logging`, nenhuma instalação adicional é necessária.
- Para `Log4j`, adicione a dependência no `pom.xml`:

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.20.0</version>
</dependency>
```

C++

- Para usar `spdlog`, instale a biblioteca via gerenciador de pacotes ou compile a partir do código fonte:

```
git clone https://github.com/gabime/spdlog.git
```

- Inclua os arquivos da biblioteca no ambiente de compilação.

Entrega

- Submeta os seguintes arquivos:
  - Código-fonte bem estruturado e comentado (disponibilizado preferencialmente no GitHub).
  - Testes unitários e de integração.
  - Vídeo (**máximo de 2 minutos**) gravado com o **Loom** (www.loom.com), demonstrando:
    - A execução dos testes.
    - O funcionamento do *logger*.
    - A depuração com *breakpoints*.
  - Relatório de no máximo 2 páginas explicando:
    - Suas escolhas técnicas.
    - O funcionamento geral do sistema.