E-fólio B | Folha de resolução para E-fólio

UNIDADE CURRICULAR: Programação por Objetos

CÓDIGO: 21093

DOCENTES: Jorge Morais, Leonel Morgado e Rúdi Gualter (tutor)

A preencher pelo estudante

NOME: Bruno Ricardo de Sá Ferreira

N.º DE ESTUDANTE: 2201529

CURSO: Engenharia Informática

DATA DE ENTREGA: 2/12/2023

Relatório do Projeto MegaSenaApp

Alterações nas Classes de Armazenamento de Dados

Conforme solicitado, procedeu-se à criação de subclasses a partir da classe DataStorageBase, visando aprimorar o armazenamento de arquivos em formatos variados e implementar conceitos avançados de programação orientada a objetos. (class DataStorageBase: ; class DataStorage(DataStorageBase): ; class DataStorageCSV(DataStorageBase):).

Subclasses de DataStorageBase

A estrutura original da DataStorageBase foi ampliada com duas subclasses: DataStorage e DataStorageCSV. Estas subclasses permitem o armazenamento de dados tanto em formatos de texto (.txt) quanto em formatos de valor separado por vírgula (CSV), respectivamente.

```
“class DataStorageCSV(DataStorageBase):
        def __init__(self, arquivo):
                super().__init__(arquivo) # chama o construtor da classe base
                if not arquivo.endswith('.csv'):
                        raise ValueError("O arquivo deve ter a extensão .csv.”)”
class DataStorage(DataStorageBase):
        def __init__(self, arquivo):
                super().__init__(arquivo)# chama o construtor da classe base
                if not (arquivo.endswith('.txt') or
arquivo.endswith('.csv')):#verifica a extensao
                        raise ValueError("O arquivo deve ter a extensão .txt
ou .csv.")
```

APLICAÇÃO DE SOBRECARGA DE MÉTODOS

Nas subclasses mencionadas, foi implementada a sobrecarga do método salvar usando o decorador @dispatch. Esta abordagem permite que o mesmo método

tenha comportamentos diferentes, dependendo do tipo de dado que está sendo processado. Por exemplo:

- **Em DataStorage**: O método salvar é sobrecarregado para lidar com listas (apostas) e combinações de inteiros e listas (resultados de sorteios), formatando e gravando estes dados em um arquivo de texto.
- **Em DataStorageCSV**: Similarmente, o método salvar é ajustado para tratar os dados de maneira apropriada para o formato CSV, facilitando a análise e o processamento futuros destes dados.

```
“@dispatch(list)#decorador para sobrecarga
def salvar(self, aposta):
        raise NotImplementedError("Este método deve ser implementado na subclasse.")#metodo deve ser implementado pelas subclasses

@dispatch(int, list)#mesma logica
def salvar(self, acertos, numeros_sorteados):
        raise NotImplementedError("Este método deve ser implementado na subclasse.”)"
```

HERANÇA E POLIMORFISMO

Estas subclasses demonstram o uso de herança, pois ambas estendem a DataStorageBase, herdando suas propriedades e estrutura básica. Ao mesmo tempo, exemplificam o polimorfismo ao implementarem de forma única o método salvar, definido na classe base.

- **Herança**: As subclasses DataStorage e DataStorageCSV herdam a estrutura fundamental da classe DataStorageBase, como a inicialização e a gestão do nome do arquivo.
- **Polimorfismo**: Cada subclasse define sua própria versão do método salvar, adaptando-o às necessidades específicas do formato de arquivo que ela gerencia (texto ou CSV).

Encapsulamento

CLASSE NUMEROCONFERENTE

- **Propósito:** A classe NumeroConferente é responsável por verificar as apostas feitas pelos usuários e calcular os prémios com base no número de acertos.
- **Encapsulamento**: A classe utiliza atributos privados para armazenar os prémios para diferentes números de acertos (4, 3, 2, 1 acerto). Implementa propriedades (getters e setters) para cada nível de prémio, protegendo os valores dos prémios de alterações diretas e garantindo que sejam atribuídos de forma controlada e segura.
- **Getters (@property):** Permitem o acesso aos valores dos prémios, possibilitando que outras partes do código consultem o valor dos prémios sem alterá-los diretamente.
- **Setters (@<property>.setter):** Estes métodos são usados para atribuir valores aos prémios. Eles incluem validações para assegurar que apenas valores inteiros positivos sejam atribuídos, garantindo a integridade dos dados.

```
@premio_4_acertos.setter #setter para premio
def premio_4_acertos(self, valor):
    if isinstance(valor, int) and valor > 0: #verifica se o valor é inteiro positivo
        self.__premio_4_acertos = valor #define valor do premio
    else:
        raise ValueError("O prêmio deve ser um número inteiro positivo.")
```

CLASSE NUMEROSSORTEADOSINTERFACE

- **Propósito**: Gerencia a interface gráfica para a exibição dos números sorteados.
- **Atributo** _aposta_feita: Um atributo privado que mantém o estado se uma aposta foi feita. "self._aposta_feita = False #atributo para verificar se uma aposta foi feita, comeca com false"

- **Getter (@property)**: Fornece um acesso seguro ao estado da aposta, permitindo que outras partes do código saibam se uma aposta foi realizada sem alterar diretamente o estado.

```
“@property
def aposta_feita(self):  #getter para aposta_feita
        return self._aposta_feita”
```

- **Setter (@<property>.setter)**: Controla a modificação do estado da aposta, incluindo validações para garantir que apenas valores booleanos sejam atribuídos, **True** or **False**, e nunca strings como “Sim” ou “Não”.

```
@aposta_feita.setter
def aposta_feita(self, valor):  #setter para aposta_feita
        if not isinstance(valor, bool):
                raise ValueError("O valor atribuído a 'aposta_feita' deve ser um booleano (True ou False).")
        self._aposta_feita = valor
```

O encapsulamento foi utilizado para proteger os dados internos das classes e para expor uma interface segura para a manipulação desses dados. Este conceito é aplicado consistentemente em todo o projeto, garantindo a integridade dos dados e facilitando a manutenção e a expansão do código.

Abstração

EXEMPLOS

Classe NumeroConferente

- **Abstração**: Esta classe abstrai o processo de conferência das apostas e cálculo dos prêmios.
- **Exemplo**: A lógica complexa de verificar os números sorteados, comparar com as apostas feitas, e calcular os prêmios é encapsulada nesta classe. Outras partes do código apenas chamam o método conferir, sem se preocupar com os detalhes internos desta operação.

```
“class NumeroConferente:
  ……
      def conferir(self, aposta, numeros_sorteados):
                acertos = len(set(numeros_sorteados) & set(aposta))
                premio = 0”
“class NumerosSorteadosInterface:
…….
       def conferir(self):
                if self.aposta_feita:  # Usa o getter para acessar aposta_feita
                        numeros_sorteados = self.gerar_sorteio()”
```

Tomadas de Decisão

USO DE CLASSES E SUBCLASSES

- **Justificação**: A estruturação do código em classes e subclasses foi adotada para promover a reutilização de código e organizar logicamente as funcionalidades do projeto. Isso facilita a manutenção e a expansão futura do aplicativo.

ENCAPSULAMENTO COM GETTERS E SETTERS

- **Justificação**: Utilizou-se getters e setters para proteger o acesso a atributos internos das classes, garantindo a integridade dos dados e escondendo detalhes internos da implementação.

ABSTRAÇÃO

- **Justificação:** A abstração foi adotada no projeto para simplificar a gestão de componentes complexos, como interfaces gráficas e operações de dados. Isso permitiu criar um código mais claro e modular, facilitando a manutenção e futuras expansões do aplicativo.

Conclusão

Neste relatório, concentrei-me nos princípios mais importantes da Programação Orientada a Objetos (POO), uma vez que a descrição detalhada das classes, métodos, objetos e outras componentes já foi abordada no relatório anterior. Este enfoque permite uma compreensão mais profunda de como as práticas e conceitos da POO foram aplicados para criar um software robusto, flexível e fácil de manter. Desta forma, realça-se a relevância destes princípios no desenvolvimento de um projeto eficiente e adaptável.