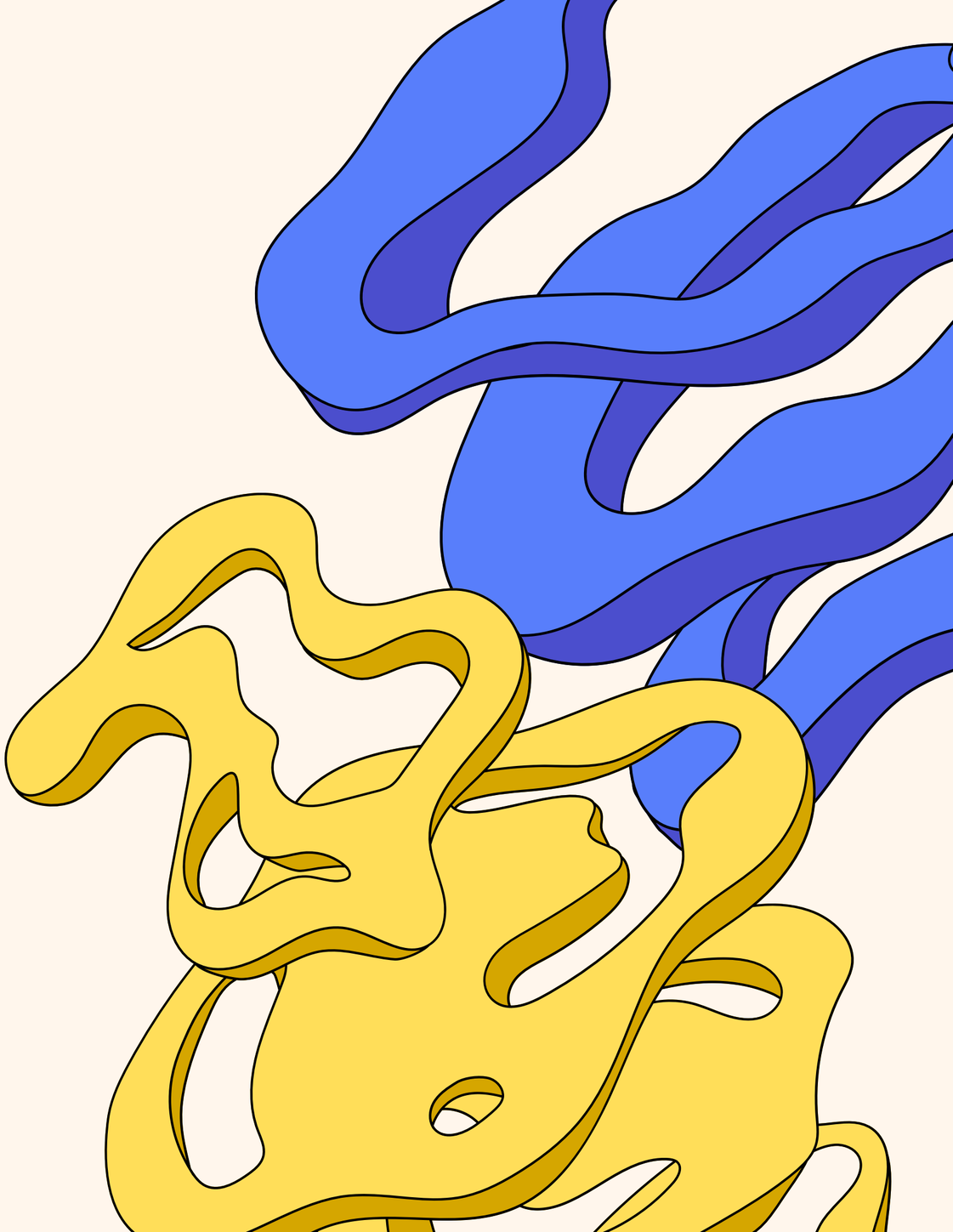
S01-Lab

POO COM PYTHON

Python Orientado a Objeto



Introduçao

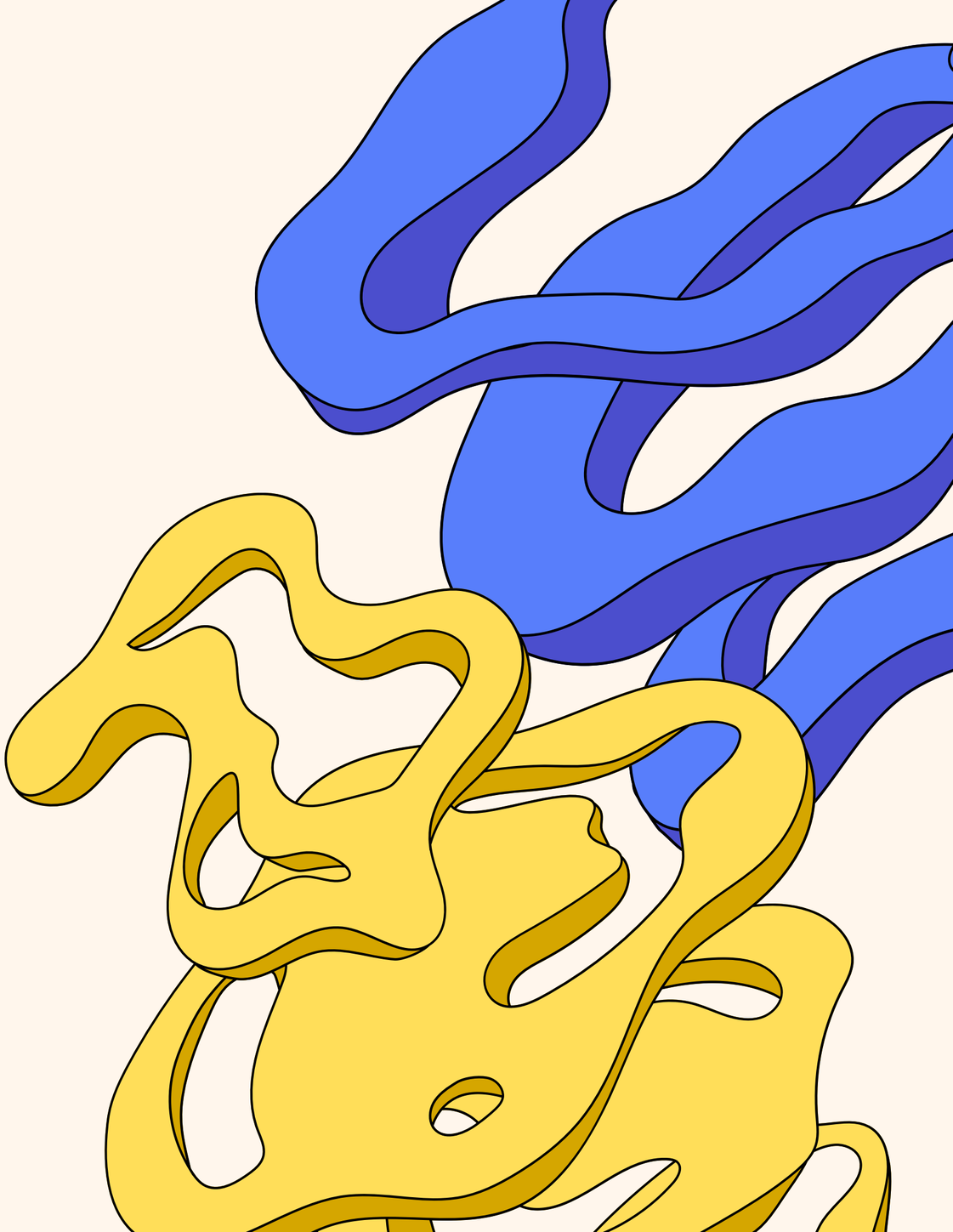
Os quatro pilares são:

- **Abstração** — representar apenas o essencial.
- **Encapsulamento** — proteger dados internos da classe.
- **Herança** — reutilizar código e comportamentos.
- **Polimorfismo** — permitir múltiplas formas de um mesmo método.

S01-Lab

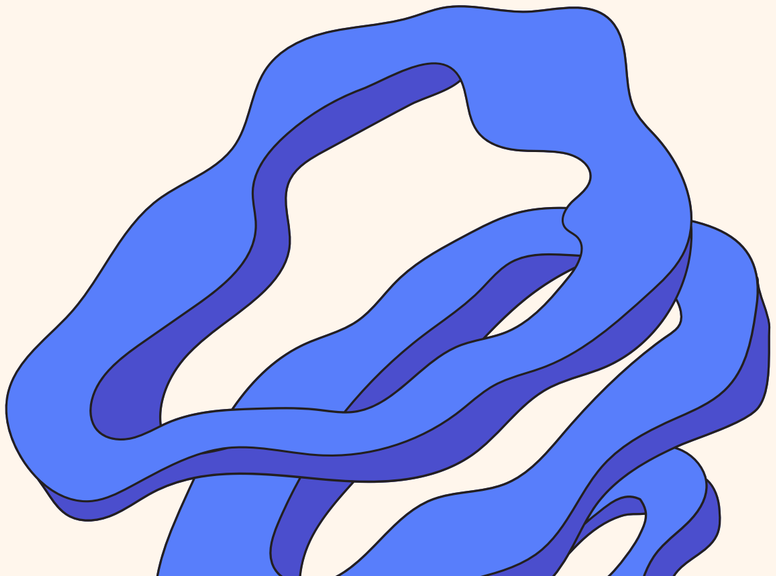
Revisão: Relações entre classes

01



S01-Lab

Agregação e composição



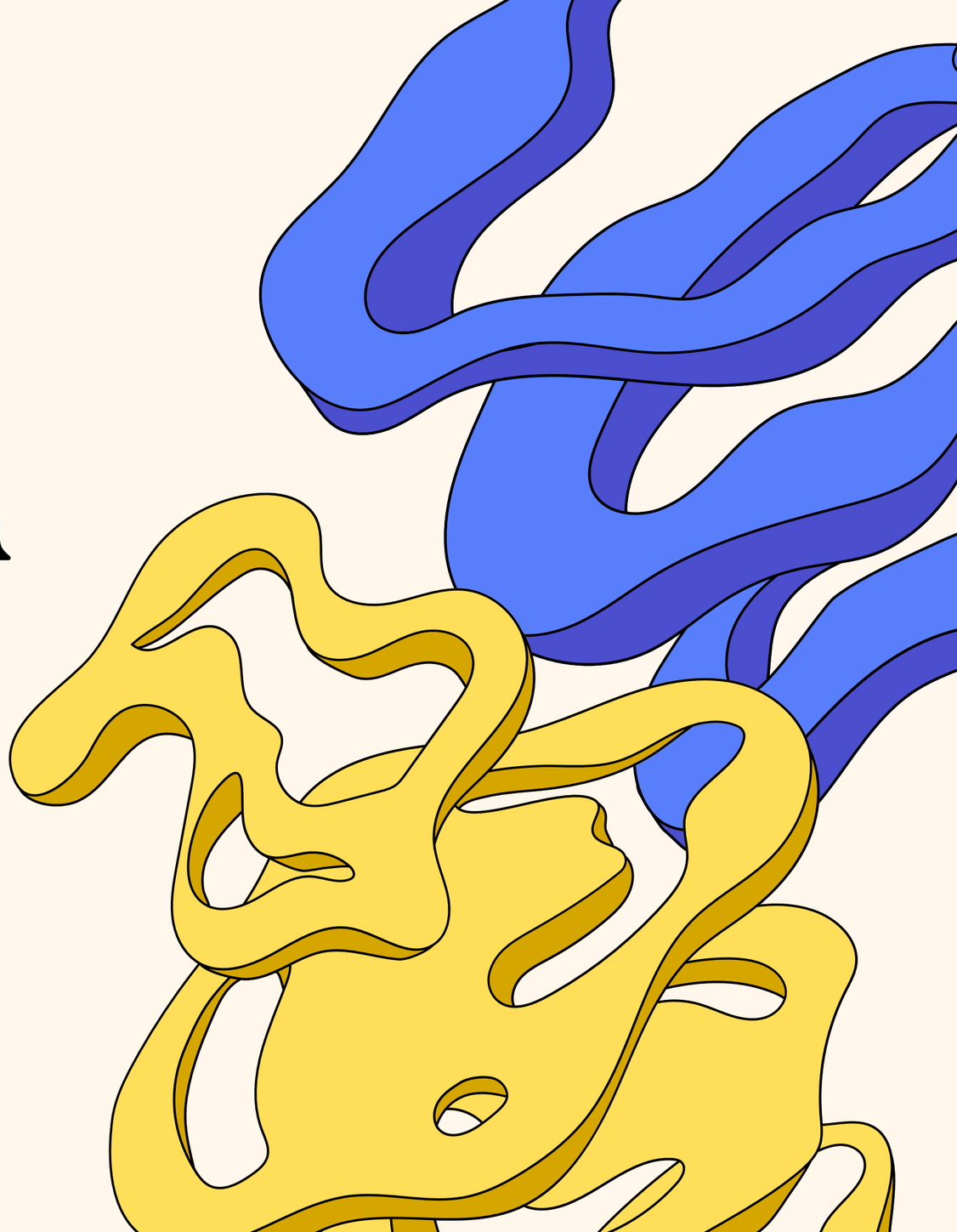
As classes podem se relacionar de duas formas principais:

- **Agregação**: Uma classe “tem” outra, mas ambas podem existir separadamente.
- Exemplo: Um Time tem Jogadores, mas um Jogador pode existir sem o Time.
- **Composição**: Uma classe é composta de outra que não existe sem ela.
- Exemplo: Um Personagem tem Equipamentos que desaparecem quando o personagem é destruído.

S01-Lab

Interfaces em Python

02



O que são Interfaces?

Interfaces definem **métodos obrigatórios** que as classes concretas devem implementar. Servem para garantir **consistência** e **flexibilidade** no código.

Em Python, usamos **classes abstratas** para representar interfaces:

Implementando Interfaces com ABC:

As interfaces em Python são criadas com o módulo abc:

Criamos uma classe que herda de ABC

Definimos métodos com o decorador @abstractmethod

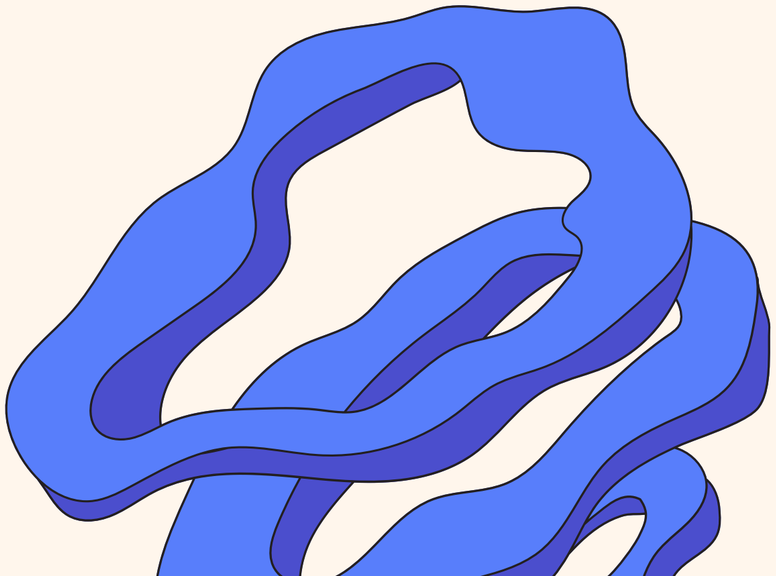
As classes concretas implementam esses métodos

```
class Guerreiro(IAcao):
def executar(self):
print(Guerreiro ataca
com espada!)
```

S01-Lab

Por que usar interfaces?

- Garantem que diferentes classes compartilhem o mesmo “contrato”.
- Facilitam a substituição de implementações.
- Tornam o código mais modular e escalável.
- Permitem o uso de **polimorfismo** de forma limpa.



Aplicando Interfaces e Relações entre classes

Nosso exemplo prático implementa um **mini-sistema de RPG** que demonstra todos os conceitos de POO em Python:

- **Interface (IAcao)** → define o contrato para métodos atacar() e defender().
- **Composição** → cada Personagem possui uma Arma.
- **Agregação** → cada Personagem tem um Inventario que gerencia uma lista de Itens.
- **Herança e Polimorfismo** → classes Guerreiro e Mago herdam de Personagem e implementam IAcao de formas diferentes.

Este exemplo conecta todos os conceitos vistos — abstração, encapsulamento, herança, polimorfismo, agregação, composição e interfaces.

S01-Lab

Exercício 1:

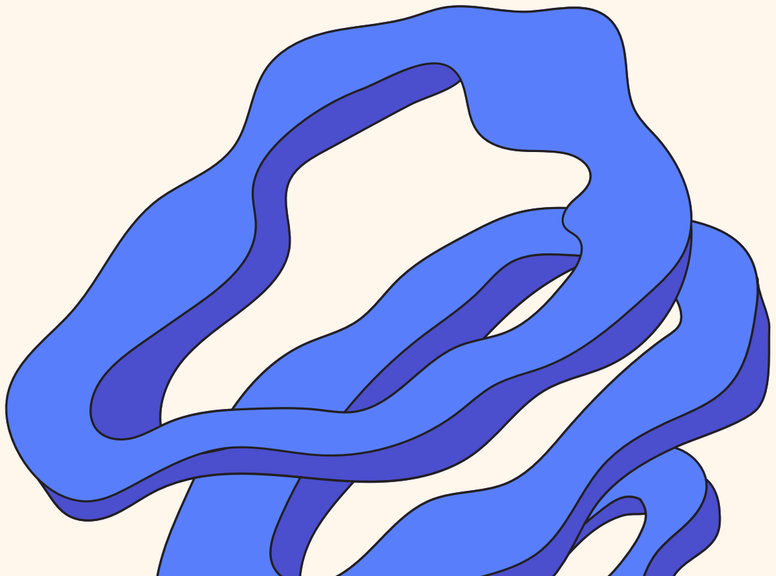
Cenário: Em Dark Souls, todos são Personagem, mas um Cavaleiro possui um atributo de ArmaduraPesada que o diferencia.

Tarefa:

1. Crie uma classe base chamada Personagem.
2. Defina os atributos _vida (inteiro) e _resistencia (inteiro) usando a convenção Python de atributos privados.
3. Implemente o método __init__ para inicializar esses atributos.
4. Use as Propriedades para criar getters e setters controlados para a vida.
5. Crie uma classe filha chamada Cavaleiro que herde de Personagem.
6. Adicione o atributo armadura_pesada (booleano) exclusivo.
7. Implemente o método __str__ em ambas as classes para retornar uma descrição formatada do objeto (ex: "Personagem com 100 de vida").

S01-Lab

EXERCÍCIO 2:



Cenário: Em Overwatch, todos os heróis são capazes de usar sua habilidade definitiva (Ultimate), mas a ação de cada um é única.

Tarefa:

1. Crie uma classe base Heroi com um método __init__ que receba o nome e a funcao (Tanque, Dano, Suporte).
2. Crie um método usar_ultimate() na classe Heroi, forçando as classes filhas a implementá-lo.
3. Crie duas classes filhas, Tanque e Dano, que herdem de Heroi.
4. Em ambas as classes filhas, sobrescreva o método usar_ultimate() para imprimir a ação específica da função.
5. No main, crie uma lista (list) e adicione instâncias de Tanque e Dano.
6. Percorra a lista e chame o método usar_ultimate() em cada herói

Exercício 3:

Cenário: O protagonista de Persona 5 (Joker) possui sua arma de combate (uma faca) e agrega membros ao seu time (Phantom Thieves).

Tarefa:

1. Crie uma classe ArmaCorpoACorpo (Composição).
2. Crie uma classe PhantomThieves. Esta será a classe agregada, dos membros da equipe do Joker. Com o __init__ recebendo nome e arma.
3. Crie a classe Joker.
    - Composição: No método __init__, instancie um objeto ArmaCorpoACorpo internamente. A arma é criada e gerenciada por Joker.
    - Agregação: O construtor de Joker deve receber uma lista (list[PhantomThieves]) de membros da equipe como parâmetro e atribuí-la a um atributo.
4. Crie um método em Joker chamado mostrar_equipe() que itera sobre a lista agregada e imprime os dados de cada membro (Adicione instâncias de PhantomThieves à sua equipe).

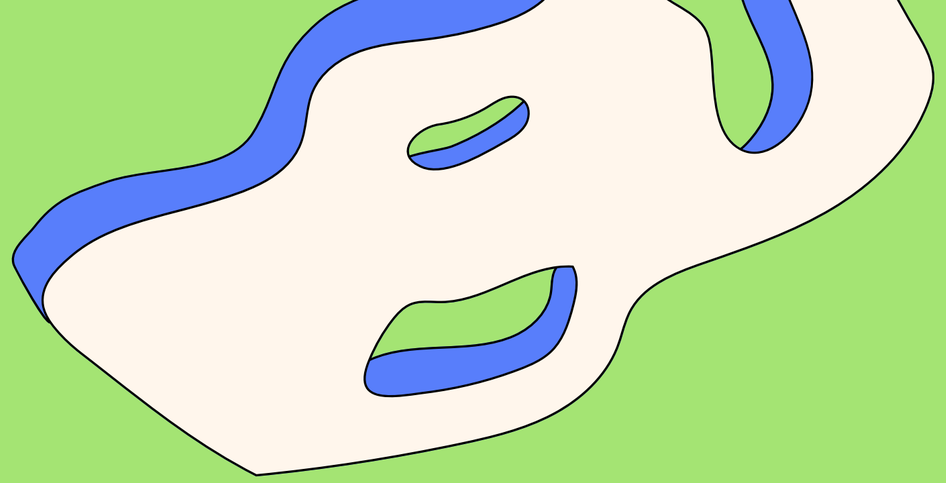
Exercício 4:

Cenário: Em Night City, muitos elementos do jogo (personagens, veículos, robôs) são Hackeaveis e Customizaveis.

Tarefa:

1. Crie uma classe Interface base chamada Cibernetico.
2. Defina um método abstrato chamado realizar_hack() nesta classe.
3. Crie uma classe Implante (Composição) com um atributo custo e funcao em seu __init__.
4. Crie a classe NetRunner que herda de Cibernetico.
    - No __init__, ele deve compor um objeto Implante
    - Ele deve implementar o método realizar_hack().
5. Crie uma classe Faccao que agregue uma lista de objetos do tipo Cibernetico em seu construtor.
6. No main, crie uma instância de Faccao e adicione múltiplos NetRunners a ela. Demonstre que a Faccao pode mandar todos os membros agregados executarem o método realizar_hack().

S01-Lab



Obrigado

zSh3rl0cK/**S01-Monitoria**

1 Contributor 0 Issues 0 Stars 0 Forks

zSh3rl0cK/S01-Monitoria

Contribute to zSh3rl0cK/S01-Monitoria development by creating an account on GitHub.

GitHub