Soluções – Teste Teórico 1 – 2024-2025

1.

- **Low Coupling**: Este princípio recomenda que os componentes de um sistema tenham o menor acoplamento possível entre si. Isso reduz a dependência entre módulos, facilitando a manutenção, reutilização e extensibilidade do sistema.
- **Information Expert**: Este princípio sugere que a responsabilidade por um determinado comportamento seja atribuída à classe que possui as informações necessárias para executá-lo. Isso melhora a coesão e a distribuição das responsabilidades no sistema.

2.

* Alterações a vermelho

```
public class UsbStorage extends Storage {
    // attributes and other methods
    public byte[] read(File fn) { /* read and return data */ }
    public boolean write(File fn, byte[] data) { /* write data in file fn */ }
}
public class CloudStorage extends Storage {
    // attributes and other methods
    public byte[] read(File fn) { /* read and return data */ }
    public boolean write(File fn, byte[] data) { /* write data in file fn */ }
}
public class TransferUtils {
    public static void copyFile(UsbStorage usb, File orig, CloudStorage ssd, File dest) {
            byte[] data = usb.read(orig);
            // work on data e.g. transform, compress, encrypt, etc…
            ssd.write(dest, data);
    }
    // other static functions...
}
public interface IStorage {
    byte[] read(File file);
    boolean write(File file, byte[] data);
}
public abstract class Storage implements IStorage {
    public abstract byte[] read(File file);
    public abstract boolean write(File file, byte[] data);
}
```

3.

a) O padrão implementado é o **Singleton**.

b) **Lazy Initialization** é uma técnica usada para atrasar a inicialização de um objeto até que ele seja necessário pela primeira vez. Isso economiza recursos e melhora a eficiência.

4.

```
public class Employee {
    private String name;
    private int department;

    private Employee() {}

    public static Builder instance() {
        return new Builder();
    }

    public static class Builder {
        private final Employee employee = new Employee();

        public Builder name(String name) {
            employee.name = name;
            return this;
        }

        public Builder department(int department) {
            employee.department = department;
            return this;
        }

        public Employee build() {
            return employee;
        }
    }
}
```

5.

a)

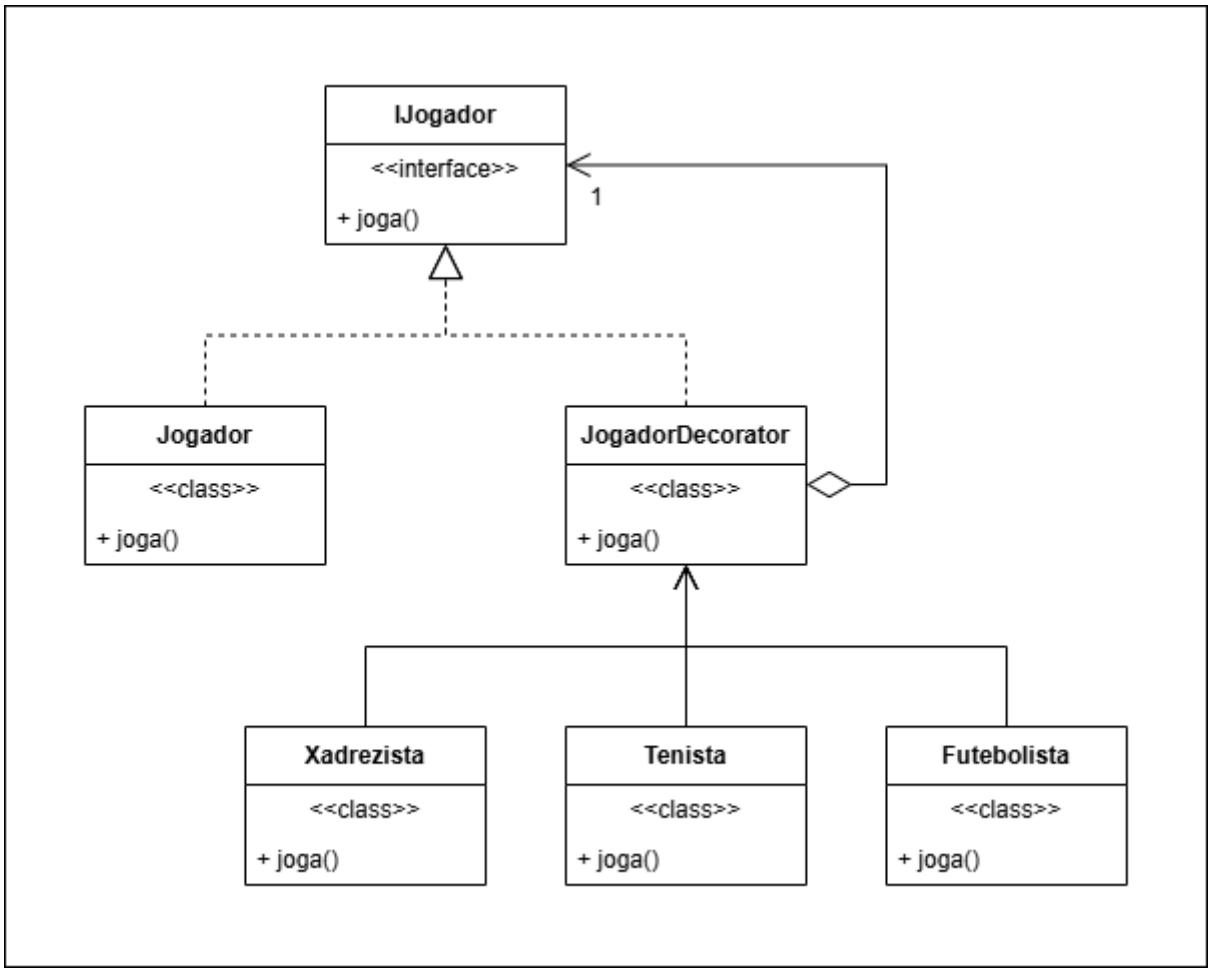
O padrão implementado é o **Adapter**.

b)

1. Implementação 1: Fusão de sistemas de bases de dados de duas empresas. Ambas suportando os mesmos métodos.

2. Implementação 2: Tipo de carregador. Um carregador Rápido e um Lento. Ambos carregam mas cada um com a velocidade que melhor suportar.

6.
Diagrama UML:



Exemplo de utilização:

```
IJogador j = new Jogador(“João”);
Tenista t = new Tenista(j1);
Xadrezista x = new Xadrezista(Tenista(new Jogador(“Maria”);
j.joga();
t.joga();
x.joga();
```

Vantagens:

- **Extensibilidade dinâmica**: Permite adicionar funcionalidades a objetos em tempo de execução sem modificar suas classes.
- **Redução da necessidade de subclasses**: Evita a criação de múltiplas subclasses para cada variação de comportamento.
- **Princípio Aberto/Fechado**: Novas funcionalidades podem ser adicionadas sem alterar o código existente.
- **Maior flexibilidade**: Decorações podem ser combinadas de diferentes formas, permitindo variações de comportamento.
- **Responsabilidade única**: Mantém classes pequenas e focadas, separando funcionalidades adicionais em decoradores distintos.