Avaliação Técnicas de Programação

Paula Nascimento Silva de Melo

QUESTOES

1) Considere os seguintes Designs Patterns e Princípios de Engenharia de Software

- Design Patterns GoF
- Design Patterns Táticos e Estratégicos - DDD
- Princípios SOLID

Explique cada um dos Designs Patterns e Princípios acima (3,0 pontos)

R:
Desing Patterns GoF: os princípios Gang of For são um dos padrões táticos que se dividem em 3 categorias:

Criacional: *Esses padrões vão definir como serão a Criação dos objetos e suas heranças resolvem problemas relacionados a escrita de classes, principalmente em como usar herança e qual a forma correta de usar.*

Exemplos: builder, factory, singleton, etc.

Estrutural: *Resolve o problema de organização do código, ele faz a unificação dos objetos e ajuda na reutilização desses.*

Exemplos: Adapter, Facade, Decorator, etc.

Comportamental: *Define como os objetos serão montados, em como será a comunicação entre esse objetos.*

Exemplos: Observer, Strategy, template method, etc.

Dentro de cada estrutura do GoF, existem várias possibilidades que se adequam com a nossa necessidade.

Design Patterns Táticos e Estratégicos - DDD:

Domain Driven Design (DDD) ,é um conjunto de práticas que ajudam na construção de um software bem projetado, o DDD refere-se à duas camadas de design:

- *estratégico: que tem por finalidade separar as responsabilidades para diminuir a complexidade do trabalho.*
- *tático: é o refinamento da código a nível de entidades, eventos, repositórios, etc. Eles auxiliam na criação de domínios.*

Princípios SOLID:
O princío SOLID ajuda a separar responsabilidade, diminuir o acoplamento e facilitar a refatoração, dividem-se em 5 princípios:

- *S: Responsabilidade Única (somente um é responsável por alterar)*
- *O: Aberto-fechado (objetos ou entidades abertos para extensão, mas fechados para modificação)*
- *L: Substituição de Liskov (A classe derivada é substituída pela classe base)*
- *I: Segregação da Interface (A classe não deve ser forçada a implementar métodos e interfaces se não vai ser útil)*
- *D: Inversão da Dependência (Está ligado em dependência entre as partes do código, dependa de abstrações e não de implementações).*

2) Indique e relacione cada um dos 22 Design Patterns em suas categorias. Use a Tabela abaixo para complementar e explicar cada um deles (3,0 pontos)

Design Pattern	Categoria	Intenção	Problema	Solução
Factory Method	Criacional	Fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados	Uma aplicação pode evoluir para uma estrutura desorganizada para a criação de objetos diferentes	Você substitui as chamadas diretas de construção de objeto (usando o operador new) por chamadas a um método fábrica especial
Builder	Criacional	Constrói objetos complexos	Quando o projeto possui objetos complexos, é necessário a inicialização de muitos campos e objetos aninhados	O padrão sugere que extraia o código de construção do objeto de sua própria classe e o mova para objetos separados chamados Builders

Design Pattern	Categoria	Intenção	Problema	Solução
Singleton	Criacional	Garante que os objetos serão criados sejam de intância única	Quando há instâncias sendo criadas em várias partes do código separadamente	Como o singleton restringe a inicilialização de um novo objeto em qualquer classe, ele garante a intância única
Abstract Factory	Criacional	Cria objetos sem especificar suas classes com maior flexibilidade	Quando precisamos combinar objetos de mesma família, é necessário atualizar os todos os objetos quando ocorre alguma atualização.	O padrão sugere declarar as interfaces para cada objeto distinto, e em seguida fazer com que todos sigam as variantes desse objeto implementando a interface
Chain of Resposability	Comportamental	Passa solicitações em uma cadeira de manipuladores	Quando temos vários componentes sequenciais e parte do código duplicada,o sistema torna-se dificil de compreender e caro de manter	O padrão sugere que vincule os manipuladores em cadeia. Cada manipulador vinculado possui um campo para armazenar uma referência ao próximo manipulador da cadeia. A solicitação percorre a cadeia até que todos os manipuladores tenham a chance de processá-la

Design Pattern	Categoria	Intenção	Problema	Solução
Command	Comportamental	Transforma uma solicitação em um objeto independente que contém todas as informações sobre esta solicitação	Quando usamos um componente que faz várias coisas, criamos várias subclasses que torna o código feio e duplicado em algumas partes	O padrão sugere que os objetos não devem enviar essas solicitações diretamente, em vez disso, devemos extrair todos os detalhes da solicitação, como o objeto que está sendo chamado, o nome do método e a lista de argumentos em uma classe de comando separada que aciona essa solicitação
Iterator	Comportamental	Percorre elementos sem expor a representação	Quando as coleções complexas fornece maneiras diferentes de acessar os elementos, você precisa acoplar o código às classes de coleção específica	O padrão permite extrair o comportamente de passagem de uma coleção em um objeto Iterator.

Design Pattern	Categoria	Intenção	Problema	Solução
Interpreter	Comportamental	Envolve a implementação de uma interface de expressão que informa para interpretar um contexto específico	Uma classe de problemas ocorre repetidamente em um domínio bem definido e compreendido. Se o domínio fosse caracterizado com um "idioma", então os problemas poderiam ser facilmente resolvidos com um "mecanismo" de interpretação	O padrão Interpreter discute: definir uma linguagem de domínio (ou seja, caracterização do problema) como uma gramática simples, representando regras de domínio como sentenças de linguagem e interpretar essas sentenças para resolver o problema
Mediator	Comportamental	Reduz dependências caóticas entre objetos	Quando a a lógica está implementada diretametne dentro do código dos objetos, torna as classes muito mais dificeis de reutilizar	O padrão sugere interromper toda a comunicação direta entre os componentes que deseja tornar independentes entre si. Em vez disso, eles devem colaborar indiretamente chamando o Mediator que redireciona as chamadas para os componentes apropriados

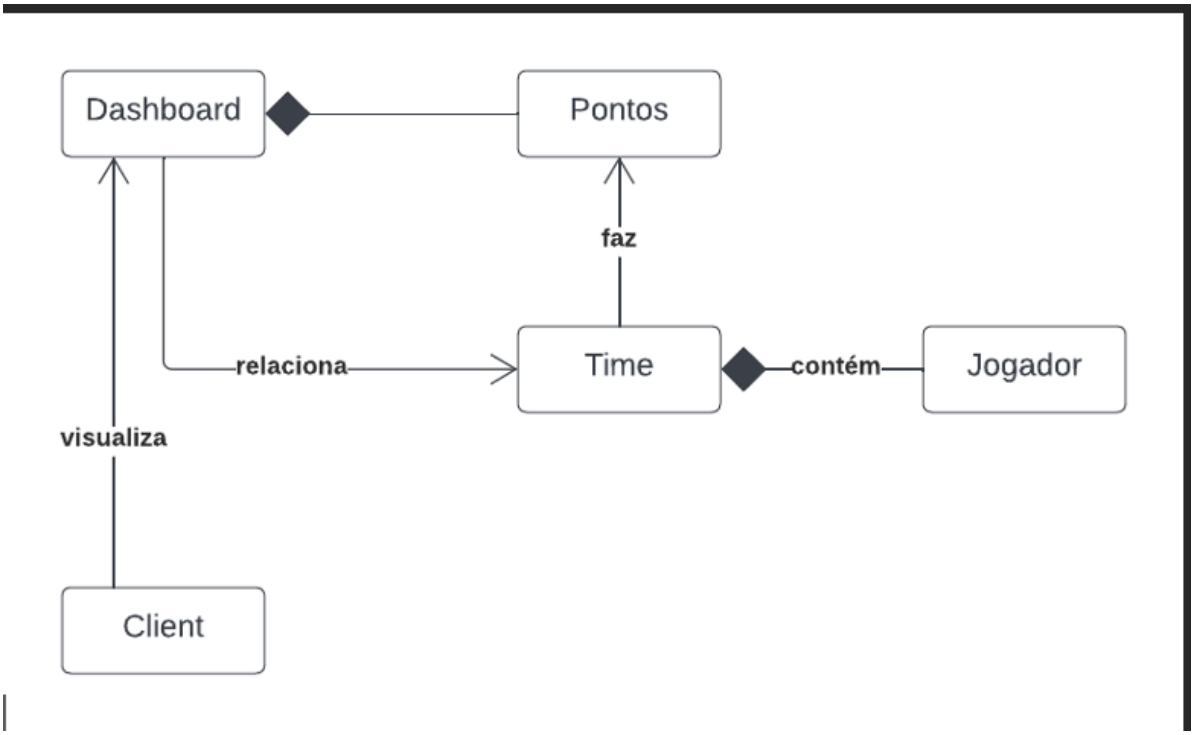
Design Pattern	Categoria	Intenção	Problema	Solução
Memento	Comportamental	Salva e restaura o estado anterior sem expor detalhes da implementação	Classes se tornam dependentes de cada pequena alteração na classe principal, o que, de outra forma, acontece em campos e métodos privados sem afetar as classes externas	O padrão delega a criação dos estados ao proprietário real desse estado. Portanto, em vez de outros objetos tentarem copiar o estado editor de fora, a própria classe do editor pode fazer, pois tem acesso total ao seu próprio estado
Observer	Comportamental	Define mecanismos de assinatura para notificar objetos sobre eventos	Quando existem objetos que são alterados, mas você quer que apenas alguns objetos saiam dessa alteração.	O padrão sugere que adicione uma assinatura à classe para que objetos individuais possam assinar ou cancelar a assinatura do fluxo de eventos
State	Comportamental	Permite que um objeto altere o comportamento quando seu estado interno for alterado	A qualquer momento há um número finito de estados que um programa pode estar,é muito difícil prever todos os estados e transições. Portanto, uma máquina de estado enxuta construída com um conjunto limitado de condicionais pode se transformar em uma bagunça inchada com o tempo	O padrão sugere que você crie novas classes para todos os estados possíveis de um objeto e extraia todos os comportamentos espeíficos do estado para essas classes

Design Pattern	Categoria	Intenção	Problema	Solução
Strategy	Comportamental	Define uma família de algoritmos e coloca em classes separadas para tornar seus objetos intercambiávei	Quando adicionamos muitos métodos dentro de uma classe, ela pode ficar enorme conflitando com o código de outras pessoas	O padrão sugere que você pegue uma classe que faz algo específico de várias maneiras diferentes e extraia todos esses algoritmos em classes separadas chamadas Strategy
Template Method	Comportamental	Define um esqueleto de algoritmo na superclasse, mas permite que as superclasses substituam etapas específicas sem alterar a estrutura	As classes possui muitos códigos semelhantes	O Padrão sugere que você divida o algoritmo em uma série de etapas, transformando essas etapas em métodos e coloque chamadas para esses métodos dentro de um único método de modelo
Visitor	Comportamental	Separa os algoritmos dos objetos nos quais eles operam	A cada alteração, é necessário mudar classes importantes de frágeis com o risco de quebrá-las	O Padrão sugere que coloque um novo comportameno em uma classe separada chamada Visitor em vez de integrá-lo às classes existentes, ajudando a executar o método adequado em um objeto sem condicionais complicadas

Design Pattern	Categoria	Intenção	Problema	Solução
Adapter	Estrutral	Permite a colaboração entre os objetos incompatívels	Quando dá problema de compatibilidade de códigos e você precisa fazer alterações que podem quebrar o código.	O Adapter permite que faça essa conversão que envolva dois objetos para que sua comunicação seja funcional
Facade	Estrutural	Fornece interface simples para um subsistema complexo	Quando precisamos trabalhar com vários objetos complexos, é necessário inicializar todos os objetos, controlar as dependências, executar os métodos um por um	O Facade fornece uma interface para trabalhar com esses objetos incluindo apenas os recursos importantes.
Bridge	Estrutural	Divide grandes classes em duas hierarquias separadas (abstração e implementação)	Adicionando novas subclasse para cada tipo, tornaria o código extenso e díficil de compreender	Permite estender as classes de forma que suas características sejam independentes
Decorator	Estrutural	Anexa novos comportamentos aos objetos	Quando criamossubclasses que combinam vários métodos, acaba por sobrecarregar o código	Permite substituir um objeto vinculado a outro alterando o comportamento em tempo de execução. Um objeto pode usar o comportamento de várias classes, tendo referências a vários objetos e delegando a eles todo tipo de trabalho

Design Pattern	Categoria	Intenção	Problema	Solução
Flyweight	Estrutural	Ajusta mais objetos á memória RAM, compartilhando partes comuns do estado entre vários objetos	Quando construímos uma aplicação grande, porém ela é muito pesada podendo não rodar por memória RAM insuficiente	O padrão sugere que pare de armazenar o estado extrínseco dentro do objeto. Em vez disso, passe esse estado para métodos específicos que dependem dele
Composite	Estrutural	Compõe objetos em estruturas de árvores para trabalhar indivualmente com cada um	Quando o nível de classes é muito aninhado torna-se desagradável a abordagem estranha ou impossível	O padrão sugere que trabalhe por meio de uma interface comum que declare os métodos necessários
Proxy	Estrutural	Fornece um subtituto para outro objeto	Quando há um objeto enorme que consome grande quantidade de recursos do sistema e precisamos usar de vez em quando	O padrão sugere que crie uma nova classe proxy com a mesma interface do objeto original. O proxy cria um objeto de serviço e delega todo o trabalho a ele

3. Considere o seguinte Diagrama UML:



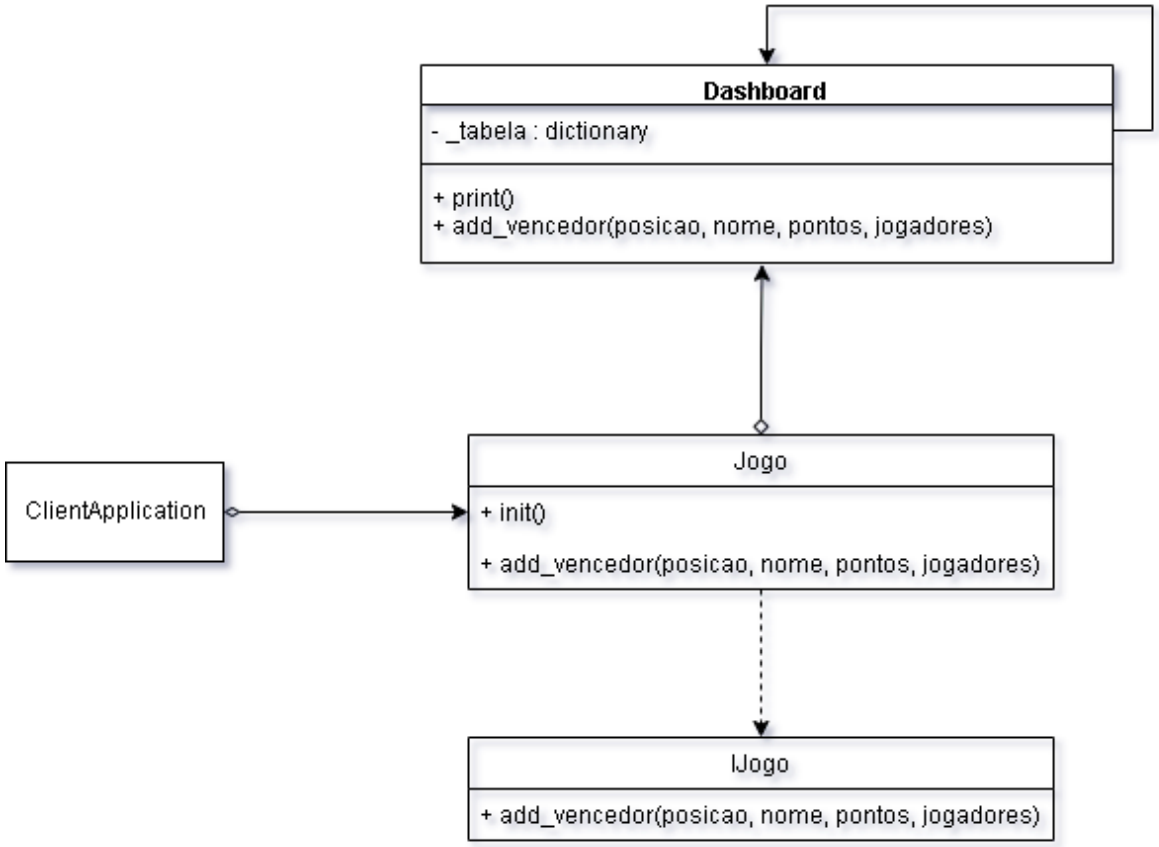
a) Crie um código em Python para representar esse Caso de Uso e aplique os Design Patterns aprendidos durante o curso - com destaque para os seguintes Design Patterns Singleton, Factory, Adapter e os princípios SOLID (3,0 pontos)

R: *O código fonte desde exercício está na pasta exercício 3 no github*

(link: https://github.com/paulademelo/tecnicas-de-programacao-Fatec-DSM/tree/main/avaliacao/exercicio3)

b) Faça um Diagrama UML da sua solução incluindo os Design Patterns aplicados nesse Caso de Uso (1,0 pontos)

R:


Dashboard
- _tabela : dictionary
+ print()
+ add_vencedor(posicao, nome, pontos, jogadores)
Jogo
+ init()
+ add_vencedor(posicao, nome, pontos, jogadores)
ClientApplication
IJogo
+ add_vencedor(posicao, nome, pontos, jogadores)