PUC Minas

Quizz 4

INFORMAÇÕES DOCENTE						
CURSO: ENGENHARIA DE SOFTWARE	**DISCIPLINA:** PROJETO DE SOFTWARE	**TURNO**	MANHÃ	TARDE	NOITE	**PERÍODO/SALA:** 4º
			x		x	
PROFESSOR (A): João Paulo Carneiro Aramuni						

Padrões GRASP – General Responsability Assignment Software Patterns

Questão 1) Considere o código abaixo:

```
class Pedido {
   private List<Item> itens;

   public double calcularTotal() {
      double total = 0;
      for (Item item : itens) {
         total += item.getPreco() * item.getQuantidade();
      }
      return total;
   }
}
```

A classe Pedido é responsável por gerenciar os itens que ela contém. No contexto de design orientado a objetos, considere os benefícios de manter o método calcularTotal na classe Pedido ao invés de delegá-lo a outra classe, como Item ou um serviço externo:

I. Delegar o cálculo a Pedido mantém a coesão alta da classe, pois a responsabilidade de calcular o total de um pedido está intrinsecamente ligada à sua coleção de itens.

II. O design é simplificado porque o cálculo é implementado diretamente onde os dados relevantes (itens) estão acessíveis e gerenciados.

III. Centraliza a lógica de negócio de cálculo do total, permitindo alterações nessa regra sem que outras partes do sistema que usam o Pedido sejam impactadas.

IV. Garante que a classe Item mantenha a Responsabilidade Única de ser um objeto de valor (ou entidade) que apenas armazena seus dados de preço e quantidade.

Quais afirmações estão corretas?

A) Apenas I e II.
B) Apenas III e IV.
C) Apenas I, II e IV.
D) Apenas II, III e IV.
E) I, II, III e IV.

Questão 2) Considere o código abaixo:

```
class Funcionario {
   private double salarioBase;
   private double bonus;

   public double calcularSalarioTotal() {
      return salarioBase + bonus;
   }
}
```

A classe Funcionario possui os dados necessários para calcular o salário total. No contexto de um design eficiente, considere os benefícios de o cálculo do salário total permanecer na classe Funcionario, em vez de ser delegado a outra classe, como Gestor ou um serviço de cálculo:

I. Centraliza a lógica de negócio do cálculo, implementando o princípio do Encapsulamento, pois as classes clientes não precisam conhecer a fórmula interna (salarioBase + bonus).

II. Reduz o acoplamento entre as classes do sistema, pois não é necessário criar uma dependência de classes externas (como um Serviço) para realizar um cálculo inerente aos dados internos de Funcionario.

III. Adere à Alta Coesão, garantindo que a classe Funcionario seja totalmente responsável pelo estado que ela controla e por todas as operações que dependem exclusivamente desse estado.

IV. Permite que as futuras alterações na regra de cálculo do salário total sejam isoladas dentro da classe Funcionario, aderindo ao Princípio do Aberto/Fechado (Open/Closed Principle).

Quais afirmações estão corretas?

A) Apenas I, II e III.
B) Apenas II e IV.
C) Apenas III e IV.
D) Apenas I e III.
E) I, II, III e IV.

Questão 3) Considere o código abaixo:

```
class Pedido {
   private List<Item> itens;

   public void adicionarItem(String nome, double preco) {
      Item item = new Item(nome, preco);
      itens.add(item);
   }
}
```

No contexto do padrão GRASP Criador (Creator), este design atribui a responsabilidade de criar instâncias de Item à classe Pedido. O Criador sugere que a classe B deve ser responsável por criar instâncias da classe A se B agrega A, contém A, usa A de perto, ou tem os dados de inicialização para A.

Analise as seguintes afirmações sobre por que Pedido é a classe mais adequada para criar Item:

I. A classe Pedido é um agregador de objetos Item, e o padrão GRASP Criador prioriza o agregador como o responsável pela criação dos objetos agregados.

II. Atribuir a responsabilidade de criação a Pedido ajuda a reduzir o acoplamento, pois elimina a necessidade de introduzir uma nova classe "Fábrica" (Factory) no design, simplificando-o.

III. A classe Pedido possui a Responsabilidade Única de criar o Item, evitando que a lógica de criação se misture com a lógica de gerenciamento de pedidos.

IV. A classe Pedido conhece os dados necessários para inicializar o Item (como nome e preco), satisfazendo um dos critérios mais fortes do padrão Criador.

Quais afirmações estão corretas?

A) Apenas I e III.
B) Apenas II e IV.
C) Apenas I e IV.
D) Apenas II e III.
E) I, II, III e IV.

Questão 4) Considere o código abaixo:

```
class Carrinho {
   private List<Produto> produtos;

   public Produto criarProduto(String nome, double preco) {
      return new Produto(nome, preco);
   }
}
```

No padrão GRASP Criador, é fundamental que a classe responsável pela criação de objetos tenha uma relação lógica ou agregação com os objetos criados.
Quais problemas poderiam surgir se a criação de `Produto` fosse movida para outra classe, como `Cliente` ou `Loja`?

A) O acoplamento entre classes não relacionadas seria aumentado, dificultando a manutenção.
B) A lógica de criação ficaria dispersa, reduzindo a clareza do design.
C) A reutilização de `Carrinho` seria prejudicada, pois ele dependeria de outras classes para criar os produtos que utiliza.
D) Todas as anteriores.
E) Nenhuma das anteriores.

Questão 5) Considere o código abaixo:

```
class Relatorio {
   private List<Dado> dados;
   public void gerarRelatorio() {
      for (Dado dado : dados) {
         System.out.println(dado.getInfo());
      }
   }
}
```

A classe `Relatorio` foi projetada para representar e manipular relatórios no sistema. No contexto do padrão GRASP Coesão Alta, é importante analisar como as responsabilidades são distribuídas entre as classes para evitar sobrecarga ou dispersão de responsabilidades.

Por que centralizar a lógica de geração de relatórios na classe `Relatorio` é mais benéfico para o design do sistema?

A) Para evitar que classes externas precisem acessar os detalhes da estrutura de dados em `Relatorio`.
B) Para permitir que a lógica de geração de relatórios seja facilmente alterada sem impactar outras partes do sistema.
C) Para garantir que a classe `Relatorio` seja o único ponto responsável por lidar com os dados que ela encapsula.
D) Todas as anteriores.
E) Nenhuma das anteriores.

Questão 6) Considere o código abaixo:

```
class Calculadora {
   public double somar(double a, double b) {
      return a + b;
   }
}
```

A classe `Calculadora` foi criada para representar operações matemáticas básicas. No contexto do padrão GRASP Coesão Alta, uma classe deve assumir responsabilidades fortemente relacionadas ao seu propósito, evitando que funcionalidades sem relação direta sejam incluídas.

Qual seria o impacto no design do sistema caso o método `somar` fosse implementado em outra classe, como `Usuario` ou `Pedido`?

A) A classe que implementasse `somar` teria responsabilidades não relacionadas, reduzindo sua coesão.
B) O design perderia clareza, dificultando a identificação de responsabilidades de cada classe.
C) A reutilização do método seria limitada, já que ele estaria em uma classe que não reflete sua função principal.
D) Todas as anteriores.
E) Nenhuma das anteriores.

Questão 7) Considere o código abaixo:

```
class Cliente {
   private Pedido pedido;

   public double obterValorTotal() {
      return pedido.calcularTotal();
   }
}
```

No padrão GRASP Acoplamento Fraco, busca-se reduzir as dependências entre classes, tornando o sistema mais flexível e fácil de manter.

Qual é a principal vantagem de delegar o cálculo do valor total ao objeto `Pedido` em vez de implementá-lo diretamente na classe `Cliente`?

A) Reduzir o acoplamento entre `Cliente` e os detalhes do cálculo no `Pedido`.
B) Manter a lógica de negócios centralizada no `Cliente` para maior controle.
C) Seguir o padrão GRASP Criador para atribuir responsabilidades.
D) Garantir que `Pedido` tenha conhecimento total da lógica de negócios do cliente.
E) Tornar o sistema dependente da estrutura interna de `Pedido`.

Questão 8)

Considere o código abaixo:

```
class Pagamento {
   private ServicoPagamento servico;

   public boolean processarPagamento(double valor) {
      return servico.autorizar(valor);
   }
}
```

No padrão GRASP Acoplamento Fraco, uma classe deve minimizar as dependências diretas com outras partes do sistema, garantindo maior flexibilidade.

Por que é vantajoso que a classe `Pagamento` delegue a lógica de autorização ao `ServicoPagamento`?

A) Para aumentar a coesão da classe `Pagamento`, mantendo-a focada no processamento.
B) Para aplicar o padrão GRASP Acoplamento Fraco e minimizar dependências diretas.
C) Para permitir que `ServicoPagamento` seja substituído ou modificado sem impactar `Pagamento`.
D) Para manter o princípio de responsabilidade única e simplificar o design.
E) Todas as anteriores.

Questão 9) Considere o código abaixo:

```
class ControladorUsuario {
   private ServicoUsuario servico;

   public boolean autenticar(String login, String senha) {
      return servico.autenticarUsuario(login, senha);
   }
}
```

O padrão GRASP Controlador sugere atribuir responsabilidades para coordenar ações do sistema a um controlador. No caso da classe ControladorUsuario. Como o princípio do padrão GRASP Controlador ajuda a manter a lógica de autenticação em conformidade com boas práticas de arquitetura?

A) Centralizando a responsabilidade de coordenação para facilitar a manutenção e escalabilidade.
B) Permitindo que detalhes técnicos de autenticação permaneçam no controlador.
C) Forçando que toda lógica de negócios fique apenas no controlador.
D) Aumentando a dependência entre o controlador e a camada de serviço.
E) Removendo a necessidade de um serviço de autenticação especializado.

Questão 10) Considere o código abaixo:

```
class ControladorVenda {
   private ServicoVenda servico;

   public boolean processarVenda(int idProduto, int quantidade) {
      return servico.registrarVenda(idProduto, quantidade);
   }
}
```

No contexto do padrão GRASP Controlador, o controlador deve atuar como um intermediário para coordenar ações entre diferentes partes do sistema. Quais benefícios de design podem ser observados ao delegar a responsabilidade de processar a venda para ControladorVenda em vez de implementar diretamente a lógica na camada de interface?

A) Facilitar a substituição ou modificação da lógica de negócios sem impactar a interface.
B) Garantir que a interface seja completamente independente da lógica de negócios.
C) Centralizar toda a lógica de negócios na camada de serviço.
D) Reduzir o número de classes no sistema para simplificar o design.
E) Promover a integração direta entre a interface e a camada de serviço.

Gabarito:

1) E - I, II, III e IV.

Atribuir o método `calcularTotal` à classe `Pedido` centraliza a lógica de cálculo, mantém o encapsulamento, evita acoplamento excessivo e facilita a manutenção. Todas as razões apresentadas são válidas.

2) E - I, II, III e IV.

Atribuir o cálculo do salário total à classe `Funcionario` reduz o acoplamento, simplifica o design, centraliza a lógica e mantém a responsabilidade onde os dados estão localizados. Todas as justificativas são pertinentes.

3) C - Apenas I e IV

Afirmações Corretas (I e IV):

> I. Correta (GRASP Criador: Agregação): O motivo mais forte para usar o Criador é a relação de agregação (Pedido agrega Item). O padrão sugere que a classe que contém ou agrega o objeto é a candidata ideal para criá-lo.
>
> IV. Correta (GRASP Criador: Dados de Inicialização): Outro critério fundamental é que Pedido recebe os dados (nome, preco) e os repassa diretamente ao construtor de Item. O Criador prioriza a classe que tem os dados necessários para iniciar a instância.

Afirmações Incorretas (II e III):

> II. Incorreta (Redução de Acoplamento): Embora simplifique o design, o ato de criar o Item dentro de Pedido aumenta o acoplamento, pois Pedido agora depende da classe Item e do seu construtor específico. A introdução de uma Fábrica (Factory) seria uma forma de reduzir o acoplamento, pois Pedido dependeria apenas da interface da Fábrica, não do construtor concreto de Item.
>
> III. Incorreta (Princípio da Responsabilidade Única): A criação do objeto (Item item = new Item(...)) é parte da responsabilidade de gerenciamento de uma coleção de itens (o que viola o Princípio da Responsabilidade Única - SRP), mas não constitui a única responsabilidade. No entanto, o padrão GRASP Criador permite essa "pequena violação" do SRP em prol do baixo acoplamento e da alta coesão global, justificando que a criação está intrinsecamente ligada à função de gerenciamento da coleção. Se o Pedido tivesse apenas a responsabilidade de criar o Item, isso seria o SRP, mas esse não é o caso; ele também gerencia a lista. O Criador está em conflito potencial com o SRP, mas é favorecido por outros padrões (como o Information Expert, que também é aplicado aqui).

4) D - Todas as anteriores.

Mover a criação de `Produto` para outra classe que não tenha relação direta com a criação dele, como `Cliente` ou `Loja`, aumentaria o acoplamento e reduziria a clareza do design. A responsabilidade deve permanecer em `Carrinho` para manter a coesão e reutilização.

5) D - Todas as anteriores.

Centralizar a lógica de geração de relatórios na classe `Relatorio` evita que outras classes acessem seus dados diretamente, facilita modificações e garante que a classe continue responsável pelos dados que encapsula, mantendo o design coeso e flexível.

6) D - Todas as anteriores.

Mover o método `somar` para uma classe como `Usuario` ou `Pedido` quebraria a coesão, dificultaria a identificação clara das responsabilidades e limitaria a reutilização. A `Calculadora` deve ser responsável por operações matemáticas básicas.

7) A - Reduzir o acoplamento entre `Cliente` e os detalhes do cálculo no `Pedido`.

Delegar o cálculo ao `Pedido` reduz a dependência da classe `Cliente` em relação ao comportamento interno do pedido, mantendo o sistema mais flexível e fácil de manter.

8) E - Todas as anteriores.

Delegar a lógica de autorização ao `ServicoPagamento` permite maior flexibilidade para substituir ou modificar o serviço de pagamento, reduz o acoplamento, aumenta a coesão da classe `Pagamento` e mantém a responsabilidade única.

9) A - Centralizando a responsabilidade de coordenação para facilitar a manutenção e escalabilidade.

O padrão GRASP Controlador ajuda a centralizar as responsabilidades de coordenação, garantindo que as mudanças possam ser feitas no controlador sem impactar diretamente a interface ou a lógica de autenticação.

10) A - Facilitar a substituição ou modificação da lógica de negócios sem impactar a interface.

Delegar a responsabilidade de processar a venda ao `ControladorVenda` permite que a interface do usuário seja independente da lógica de negócios, facilitando a manutenção e a substituição de partes do sistema sem afetar a interface.