LISTA DE EXERCÍCIOS 2

Total de pontos 17/20

Arquitetura de Sofware e Cloud Computing

O e-mail do participante (**athosgustavowf@gmail.com**) foi registrado durante o envio deste formulário.

Nome *

athos

Matrícula *

01561587

Curso *

Análise e desenvolvimento de sistemas

Turma *

5NA-EMB

Para as questões a seguir, assinale apenas as afirmativas corretas.

No contexto de Diagramas de Contexto Arquitetural: * 1/1

- [ ] 1. O diagrama de contexto é usado para modelar a estrutura interna de um sistema.
- [ ] 2. Os atores em um diagrama de contexto são sempre sistemas de software.
- [x] 3. O diagrama de contexto ajuda a identificar as interfaces entre o sistema e o ambiente externo.
- [ ] 4. Diagrama de contexto nunca inclui fluxos de dados.
- [x] 5. O diagrama de contexto mostra a interação entre o sistema e seus atores externos.

Em relação aos Elementos do Diagrama de Contexto Arquitetural: * 1/1

- [ ] 1. Atores são elementos internos do sistema.
- [ ] 2. Os fluxos de dados não são representados em diagramas de contexto.
- [x] 3. O diagrama de contexto pode incluir sistemas externos como atores.
- [ ] 4. Os atores são apenas pessoas.
- [x] 5. O diagrama de contexto foca na interação do sistema com elementos externos.

Sobre um Exemplo de Projeto de E-commerce: * 1/1

- [x] 1. O gateway de pagamento é considerado um ator no diagrama de contexto.
- [ ] 2. O serviço de entrega não interage diretamente com o sistema.
- [x] 3. Usuários finais são os principais atores no sistema de e-commerce.
- [ ] 4. O sistema de marketing pode ser considerado um componente interno.
- [x] 5. O visitante do site é um ator distinto do usuário final.

No contexto de Arquitetura baseada em Componentes: * 1/1

- [x] 1. Componentes são unidades autônomas de software com interfaces bem definidas.
- [ ] 2. Componentes não podem ser reutilizados em diferentes sistemas.
- [x] 3. Componentes comunicam-se através de interfaces.
- [ ] 4. Cada componente deve ser independente de todos os outros.
- [ ] 5. Componentes são sempre desenvolvidos internamente.

Referente a O que é um Componente?: * 1/1

- [x] 1. Um componente é uma unidade funcional de software.
- [ ] 2. Um componente não precisa ter interfaces públicas.
- [x] 3. Componentes podem ser compostos por outros componentes.
- [ ] 4. Componentes são sempre desenvolvidos a partir do zero.
- [x] 5. A comunicação entre componentes ocorre exclusivamente por meio de suas interfaces.

Sobre as Etapas do projeto de um Componente: * 1/1

- [x] 1. A análise de requisitos é a primeira etapa.
- [ ] 2. A implementação do componente ocorre antes do design.
- [x] 3. A validação do componente ocorre após a implementação.
- [ ] 4. O design de interfaces não é considerado na etapa de design.
- [x] 5. Componentes devem ser testados isoladamente antes da integração.

Referente à Reutilização de Componentes: * 0/1

- [x] 1. Componentes reutilizáveis devem ser genéricos o suficiente para vários contextos.
- [x] 2. A reutilização de componentes reduz o custo de desenvolvimento.
- [ ] 3. Componentes reutilizáveis não devem ser documentados para facilitar a adaptação.
- [ ] 4. Componentes especializados são mais reutilizáveis.
- [ ] 5. A integração de componentes reutilizáveis requer adaptação mínima.

Sobre Interface de Arquitetura de Sistemas: * 1/1

- [x] 1. A interface define como os componentes interagem entre si.
- [ ] 2. Interfaces devem ser complexas para evitar reutilização indevida.
- [ ] 3. A documentação da interface é opcional.
- [ ] 4. A interface pode ser adaptada durante a fase de implementação.
- [x] 5. Interfaces devem ser estáveis e bem definidas.

Sobre Análise e Modelagem de Interfaces: * 1/1

- [x] 1. A análise de interfaces é essencial para definir as interações entre componentes.
- [ ] 2. A modelagem de interfaces só deve ocorrer após a implementação do componente.
- [ ] 3. Interfaces não devem ser testadas isoladamente.
- [x] 4. A modelagem de interfaces ajuda a prever problemas de integração.
- [x] 5. Interfaces devem ser flexíveis para adaptação futura.

Referente à Validação de Interfaces: * 1/1

- [x] 1. A validação garante que a interface atenda aos requisitos de funcionalidade.
- [ ] 2. Interfaces não precisam ser validadas isoladamente.
- [ ] 3. A validação ocorre após a fase de teste do sistema.
- [ ] 4. A validação é uma etapa opcional no desenvolvimento.
- [x] 5. Interfaces validadas são menos propensas a falhas de integração.

Sobre Identificação do Usuário, das Tarefas e dos Requisitos do Ambiente: * 1/1

- [x] 1. A identificação do usuário é a primeira etapa na definição de requisitos.
- [ ] 2. As tarefas dos usuários não influenciam o design de interação.
- [x] 3. Requisitos do ambiente incluem as restrições de hardware e software.
- [ ] 4. A identificação dos requisitos do ambiente é desnecessária em sistemas genéricos.
- [x] 5. A análise das tarefas dos usuários ajuda a definir funcionalidades do sistema.

Sobre o Design de Interação: * 1/1

- [x] 1. O design de interação foca na usabilidade do sistema.
- [ ] 2. A interação com o usuário deve ser o mais automatizada possível, sem considerações de usabilidade.
- [x] 3. Design de interação inclui a definição de fluxos de trabalho do usuário.
- [ ] 4. O design de interação deve ignorar as limitações do hardware.
- [x] 5. O design de interação deve considerar a experiência do usuário.

Sobre Padrões de Projeto: * 1/1

- [x] 1. Padrões de projeto fornecem soluções reutilizáveis para problemas recorrentes.
- [ ] 2. Os padrões de projeto são específicos para cada linguagem de programação.
- [x] 3. Padrões de projeto são uma forma de promover boas práticas de desenvolvimento.
- [ ] 4. Padrões de projeto não são aplicáveis em sistemas modernos.
- [x] 5. A utilização de padrões de projeto pode simplificar a manutenção do sistema.

Sobre as Vantagens dos Padrões de Projeto: * 1/1

- [x] 1. Padrões de projeto ajudam a reduzir a complexidade do código.
- [ ] 2. Padrões de projeto aumentam o custo de desenvolvimento.
- [x] 3. Padrões de projeto facilitam a comunicação entre desenvolvedores.
- [ ] 4. Padrões de projeto são difíceis de adaptar a novas tecnologias.
- [x] 5. Padrões de projeto promovem a consistência do código.

Sobre os Padrões GoF: * 0/1

- [x] 1. Padrões GoF incluem padrões de criação, estruturais e comportamentais.
- [x] 2. O padrão Singleton é um exemplo de padrão estrutural.
- [x] 3. O padrão Factory Method é um exemplo de padrão de criação.
- [x] 4. O padrão Observer é usado para definir um relacionamento de um-para-muitos.
- [ ] 5. O padrão Adapter converte a interface de uma classe em outra interface esperada pelos clientes.

Sobre os Padrões de Criação (Creational Patterns): * 0/1

- [x] 1. Padrões de criação tratam da forma de instanciar objetos.
- [x] 2. O padrão Prototype permite a criação de novos objetos clonando instâncias existentes.
- [ ] 3. O padrão Abstract Factory fornece uma interface para criar uma família de objetos relacionados.
- [x] 4. O padrão Builder é usado para separar a construção de um objeto complexo de sua representação.
- [ ] 5. O padrão Singleton permite a criação de múltiplas instâncias de uma classe.

Sobre os Padrões Estruturais (Structural Patterns): * 1/1

- [x] 1. Padrões estruturais tratam da composição de classes e objetos.
- [x] 2. O padrão Composite permite que objetos individuais e composições de objetos sejam tratados de forma uniforme.
- [x] 3. O padrão Adapter converte a interface de uma classe em outra interface esperada pelos clientes.
- [x] 4. O padrão Decorator permite a adição de funcionalidades a um objeto dinamicamente.
- [ ] 5. O padrão Bridge promove o acoplamento forte entre uma abstração e sua implementação.

Sobre os Padrões Comportamentais (Behavioral Patterns): * 1/1

- [x] 1. Padrões comportamentais lidam com a interação e responsabilidade entre objetos.
- [x] 2. O padrão Observer define uma dependência um-para-muitos entre objetos.
- [x] 3. O padrão Strategy permite que uma família de algoritmos seja intercambiável.
- [x] 4. O padrão Command encapsula uma solicitação como um objeto, permitindo parametrizar clientes com filas ou solicitações.
- [ ] 5. O padrão State não permite que um objeto altere seu comportamento quando seu estado interno muda.

Sobre a Origem do Reúso de Software: * 1/1

- [x] 1. O reúso de software surgiu como uma prática para aumentar a eficiência do desenvolvimento de software.
- [ ] 2. Reúso de software sempre foi uma prática comum desde os primórdios da engenharia de software.
- [x] 3. A prática de reúso de software é incentivada para reduzir a redundância no código.
- [x] 4. O reúso de software se tornou mais formalizado com o advento dos padrões de projeto.
- [ ] 5. O reúso de software é um conceito relativamente novo na indústria de software.

Sobre Tipos e Técnicas de Reúso de Software: * 1/1

- [x] 1. O reúso de código é uma técnica comum de reúso de software.
- [x] 2. Reúso de software pode ocorrer por meio de componentes, frameworks e bibliotecas.
- [ ] 3. O reúso de software é limitado a linguagens de programação específicas.
- [ ] 4. O reúso de software não abrange o reúso de processos de desenvolvimento.
- [x] 5. Técnicas de reúso podem incluir a customização de componentes genéricos para aplicações específicas.