INSTITUTO SUPERIOR POLITÉCNICO DE TECNOLOGIA E CIÊNCIAS

DEPARTAMENTO DE ENGENHARIA E TECNOLOGIA

CURSO: ENGENHARIA INFORMÁTICA

ENGENHARIA DE SOFTWARE II

RELATÓRIO

ELEMENTOS DO GRUPO

- **ONÉSIMO MARTINS – 20201076**
- **HELPIDIO MATEUS – 20201608**
- **EUGÉNIO CHIMUCO – 20200787**

Turma: EINFM

1. Desenho e Implementação do Software:

o Quais são os princípios fundamentais de design de software e como eles influenciam a qualidade do código?

R:

1. Princípio da responsabilidade única (Single Responsibility Principle - SRP): Cada classe ou módulo deve ter uma única responsabilidade. Isso ajuda a manter o código coeso, facilitando a compreensão e a manutenção.
2. Princípio aberto/fechado (Open/Closed Principle - OCP): O código deve ser aberto para extensão, mas fechado para modificação. Isso significa que você pode adicionar novos recursos ou comportamentos sem precisar modificar o código existente.
3. Princípio de substituição de Liskov (Liskov Substitution Principle - LSP): As classes derivadas devem ser substituíveis por suas classes base sem afetar a corretude do programa. Isso garante que as subclasses possam ser usadas no lugar de suas classes base sem causar problemas.
4. Princípio de segregação de interface (Interface Segregation Principle - ISP): As interfaces devem ser específicas para os clientes que as utilizam. Em vez de ter interfaces grandes e genéricas, é melhor ter interfaces menores e mais especializadas, adaptadas às necessidades de cada cliente.
5. Princípio de inversão de dependência (Dependency Inversion Principle - DIP): Os módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem

depender de abstrações. Isso ajuda a criar um acoplamento fraco entre os componentes do sistema, facilitando a modificação e a substituição de partes do código.

6. Princípio de interface limpa (Don't Repeat Yourself - DRY): Evite duplicação de código, pois isso aumenta a complexidade e a manutenção do sistema. Em vez disso, procure reutilizar código existente por meio de abstrações e componentes compartilhados.

Esses princípios influenciam a qualidade do código de várias maneiras:

- Melhor organização: Eles ajudam a estruturar o código de maneira clara e lógica, tornando-o mais fácil de entender e dar manutenção.

- Flexibilidade: Os princípios promovem um design flexível e extensível, permitindo que o software se adapte às mudanças e evoluções dos requisitos.

- Reutilização: Ao seguir os princípios de design, é mais provável que você crie componentes reutilizáveis, reduzindo a duplicação de código e aumentando a eficiência do desenvolvimento.

- Testabilidade: O código bem projetado com base nesses princípios é mais fácil de testar, pois as responsabilidades são bem definidas e as dependências são gerenciadas adequadamente.

- Manutenibilidade: O código que segue esses princípios é mais fácil de manter, pois é mais modular e menos propenso a introduzir erros durante as alterações.

- Escalabilidade: Os princípios de design promovem um código mais escalável, permitindo que o software cresça e

se adapte a demandas crescentes sem grandes modificações.

o Quais são os padrões de projeto de software mais comuns e em quais situações eles podem ser aplicados?
R:
1. Padrões de Criação:
- Singleton: Utilizado quando é necessário garantir que uma classe tenha apenas uma instância e fornecer um ponto de acesso global para essa instância.
- Factory Method: Utilizado quando há a necessidade de criar objetos de diferentes classes derivadas de uma mesma classe base, delegando a responsabilidade de criação para as subclasses.

2. Padrões Estruturais:

- Adapter: Utilizado para conectar interfaces incompatíveis, permitindo que classes com interfaces diferentes trabalhem juntas.
- Decorator: Utilizado para adicionar responsabilidades adicionais a um objeto dinamicamente, sem modificar sua estrutura básica.
- Composite: Utilizado para tratar objetos individuais e composições de objetos de maneira uniforme, permitindo que clientes tratem objetos individuais e grupos de objetos de forma transparente.

3. Padrões Comportamentais:

- Observer: Utilizado para estabelecer uma relação de dependência entre objetos, de modo que quando um objeto muda de estado, todos os objetos dependentes são notificados e atualizados automaticamente.
- Strategy: Utilizado quando se deseja ter diferentes algoritmos disponíveis para executar uma tarefa e poder selecionar o algoritmo a ser usado em tempo de execução.
- Command: Utilizado para encapsular uma solicitação como um objeto, permitindo que os clientes parametrizem os objetos com diferentes solicitações e que essas solicitações possam ser enfileiradas ou registradas.

4. Padrões Arquiteturais:
- MVC (Model-View-Controller): Utilizado para separar a lógica de negócios (Model), a apresentação (View) e a interação do usuário (Controller) em componentes distintos.
- MVP (Model-View-Presenter): Uma variação do padrão MVC que enfatiza uma separação clara entre a lógica de apresentação (Presenter) e a interface do usuário (View).
- MVVM (Model-View-ViewModel): Um padrão que separa a lógica de negócios (Model) da lógica de apresentação (ViewModel) e da interface do usuário (View), com a utilização de vinculação de dados (data binding).

o Como vocês podem garantir a qualidade do software durante o processo de desenvolvimento?
R:

1. Planejamento adequado: Inclui a definição clara dos requisitos, escopo e metas do projeto. Isso ajuda a estabelecer expectativas realistas e a criar uma base sólida para o desenvolvimento.

2. Design de software sólido: Um bom design de software, seguindo princípios e padrões de design adequados, ajuda a criar uma base estrutural sólida para o código. Isso facilita a manutenção, extensibilidade e testabilidade do sistema.

3. Revisões de código: Realizar revisões de código por outros membros da equipe ajuda a identificar problemas e melhorar a qualidade do código. Essas revisões podem identificar erros, verificar a adesão a padrões de codificação, avaliar a legibilidade e fornecer sugestões para melhorias.

4. Testes automatizados: A automação de testes é fundamental para verificar a funcionalidade e a estabilidade do software. Isso inclui testes unitários, testes de integração, testes de sistema e testes de aceitação. Ter uma suíte abrangente de testes automatizados ajuda a identificar problemas rapidamente e garante que as alterações futuras não introduzam regressões.

5. Integração Contínua e Entrega Contínua (CI/CD): A prática de integração contínua envolve a integração frequente do código no repositório principal, seguida de compilação e execução de testes automatizados. A entrega contínua vai além disso, envolvendo a automatização do processo de implantação do software em ambientes de produção. Essas práticas ajudam a identificar problemas mais cedo e a fornecer um fluxo de entrega mais rápido e confiável.

6. Gerenciamento de problemas e rastreamento de bugs: Ter um sistema de gerenciamento de problemas e um processo eficaz de rastreamento de bugs ajuda a registrar, priorizar e resolver problemas identificados durante o desenvolvimento e após a entrega do software.

7. Monitoramento e feedback dos usuários: Incorporar recursos de monitoramento no software permite coletar dados sobre o desempenho, erros e uso do sistema em tempo real. Além disso, coletar feedback dos usuários e incorporar melhorias com base nesse feedback ajuda a melhorar continuamente a qualidade do software.

8. Treinamento e capacitação da equipe: Investir no treinamento e capacitação da equipe em boas práticas de desenvolvimento de software, padrões de design, testes automatizados e outras técnicas relevantes é essencial para garantir a qualidade do software.

2. Escolha da Arquitetura do Software:
o Quais são os diferentes tipos de arquiteturas de software e quais são suas características distintas?
R:
1. Arquitetura em camadas (Layered Architecture):
A arquitetura em camadas organiza o sistema em camadas distintas, onde cada camada possui responsabilidades bem definidas. As camadas são empilhadas uma sobre a outra, e a comunicação ocorre geralmente de cima para baixo. Esse tipo de arquitetura promove a separação de preocupações e facilita a manutenção e a modificação do sistema.

2. Arquitetura Cliente-Servidor (Client-Server Architecture):

Na arquitetura cliente-servidor, o sistema é dividido em duas partes principais: o cliente, que solicita serviços, e o servidor, que fornece esses serviços. Os clientes enviam solicitações ao servidor e recebem as respostas correspondentes. Essa arquitetura permite a distribuição de tarefas entre os clientes e os servidores, facilitando a escalabilidade e a manutenção do sistema.

3. Arquitetura de Microserviços (Microservices Architecture):
Na arquitetura de microserviços, o sistema é dividido em pequenos serviços independentes, cada um responsável por uma função específica. Cada serviço é executado em seu próprio processo e pode ser implantado, dimensionado e atualizado independentemente. Essa abordagem facilita a escalabilidade, a manutenção e a flexibilidade do sistema.

4. Arquitetura Orientada a Eventos (Event-Driven Architecture):
Na arquitetura orientada a eventos, os componentes do sistema comunicam-se por meio da produção e consumo de eventos. Os eventos são emitidos quando algo significativo acontece no sistema, e os componentes interessados respondem a esses eventos. Essa abordagem permite a criação de sistemas assíncronos, escaláveis e desacoplados.

5. Arquitetura em Pipeline (Pipeline Architecture):
A arquitetura em pipeline é usada quando há uma sequência de etapas a serem executadas em um sistema. Cada etapa recebe um conjunto de dados de entrada, processa-os e passa os

resultados para a próxima etapa. Essa arquitetura é comumente encontrada em sistemas de processamento de dados, como pipelines de ETL (Extração, Transformação e Carga).

6. Arquitetura baseada em Componentes (Component-Based Architecture):
Na arquitetura baseada em componentes, o sistema é construído por meio da composição de componentes reutilizáveis. Cada componente encapsula uma funcionalidade específica e pode ser usado em diferentes contextos. Essa abordagem promove a reutilização de código e facilita a modularidade e a manutenção do sistema.
o Como vocês podem determinar qual arquitetura é mais adequada para um determinado projeto?
R:
1. Compreender os requisitos: Analise os requisitos funcionais e não funcionais do projeto, incluindo escalabilidade, desempenho, segurança, manutenção, integração com outros sistemas, entre outros. Identifique as principais necessidades e restrições do projeto.

2. Avaliar o contexto do projeto: Considere o contexto no qual o projeto está inserido, como o ambiente operacional, o tamanho da equipe de desenvolvimento, o prazo do projeto e os recursos disponíveis. Isso pode influenciar a escolha da arquitetura.

3. Analisar padrões e melhores práticas: Estude os padrões de arquitetura de software e as melhores práticas adotadas na indústria para projetos semelhantes. Isso pode fornecer insights valiosos sobre as abordagens comprovadas em casos semelhantes.

4. Avaliar as características do projeto: Considere as características específicas do projeto, como complexidade, volume de dados, requisitos de tempo real, integração com sistemas legados, capacidade de evolução e flexibilidade. Essas características podem influenciar a escolha da arquitetura mais apropriada.

5. Prototipar e iterar: Em alguns casos, pode ser útil criar protótipos ou implementações iniciais para validar diferentes abordagens arquiteturais. Essa abordagem permite avaliar a eficácia de cada arquitetura em relação aos requisitos e fazer ajustes conforme necessário.

6. Considerar o feedback da equipe: Inclua a equipe de desenvolvimento no processo de seleção da arquitetura. Os desenvolvedores podem fornecer insights valiosos com base em sua experiência e conhecimento técnico.

7. Realizar análise de riscos: Considere os riscos associados a cada arquitetura em potencial. Avalie os possíveis impactos negativos e os esforços necessários para mitigar esses riscos.

o Quais são os fatores que vocês devem considerar ao escolher uma arquitetura, como desempenho, escalabilidade e segurança?

R:

1. Desempenho:

- Requisitos de tempo de resposta: Determine as necessidades de tempo de resposta do sistema e avalie se a arquitetura proposta pode atender a esses requisitos.
- Processamento paralelo: Considere se a arquitetura permite a execução paralela de tarefas e aproveitamento eficiente dos recursos de hardware.
- Uso eficiente de recursos: Avalie se a arquitetura é capaz de utilizar eficientemente os recursos disponíveis, como CPU, memória e rede.

2. Escalabilidade:

- Escalabilidade horizontal e vertical: Verifique se a arquitetura suporta escalabilidade horizontal (adicionando mais instâncias do sistema) e vertical (aumentando a capacidade de uma única instância).
- Distribuição de carga: Considere se a arquitetura permite distribuir a carga de trabalho de forma equilibrada entre os componentes do sistema, para garantir que ele possa lidar com um aumento na demanda.
- Gerenciamento de recursos: Avalie se a arquitetura facilita o gerenciamento eficiente de recursos, como balanceamento de carga, compartilhamento de recursos e gerenciamento de cache.

3. Segurança:

- Controle de acesso: Verifique se a arquitetura fornece mecanismos para controlar e autenticar o acesso aos recursos do sistema, garantindo que apenas usuários autorizados possam interagir com o sistema.
- Proteção de dados: Considere se a arquitetura oferece recursos para proteger os dados em trânsito e em repouso, como criptografia e mecanismos de segurança adequados.
- Resiliência a ataques: Avalie se a arquitetura é projetada para resistir a ataques comuns, como injeção de código, ataques de negação de serviço (DDoS) e exploração de vulnerabilidades conhecidas.

4. Manutenibilidade:

- Facilidade de modificação: Considere se a arquitetura permite a fácil adição, remoção ou modificação de componentes do sistema sem afetar adversamente outros componentes.
- Separação de preocupações: Verifique se a arquitetura promove a separação clara das responsabilidades entre os diferentes componentes, facilitando a manutenção e a evolução do sistema.
- Testabilidade: Avalie se a arquitetura facilita a criação e a execução de testes automatizados, permitindo uma cobertura abrangente dos diferentes componentes do sistema.