1. **Os artigos foram uteis?**
2. **Os artigos conseguiram transmitir os conceitos teóricos com clareza?**
3. **Possuía conhecimento sobre os assuntos anteriormente?**

4. **O que é SOLID?**
   SOLID é uma tese criada por Robert C. Martin (Uncle Bob), que dita cinco princípios da orientação a objetos e design de código.

5. **Qual são os benefícios ao se aplicar SOLID?**
   Esses princípios ajudam o programador a escrever códigos mais limpos, separando responsabilidades, diminuindo acoplamentos, facilitando na refatoração e estimulando o reaproveitamento do código.

6. **Descreva: Single Responsiblity Principle (Princípio da responsabilidade única)**
   Uma classe deve ter um, e somente um, motivo para mudar.
   Esse princípio declara que uma classe deve ser especializada em um único assunto e possuir apenas uma responsabilidade dentro do software, ou seja, a classe deve ter uma única tarefa ou ação para executar.

7. **Descreva: Open-Closed Principle (Princípio Aberto-Fechado)**
   Objetos ou entidades devem estar abertos para extensão, mas fechados para modificação, ou seja, quando novos comportamentos e recursos precisam ser adicionados no software, devemos estender e não alterar o código fonte original.
   Sua principal vantagem é a facilidade na adição de novos requisitos, diminuindo as chances de introduzir novos bugs pois o novo comportamento fica isolado, e o que estava funcionando provavelmente continuara funcionando.

8. **Descreva: Liskov Substitution Principle (Princípio da substituição de Liskov)**
   Se S é um subtipo de T, então os objetos do tipo T, em um programa, podem ser substituídos pelos objetos de tipo S sem que seja necessário alterar as propriedades deste programa.
   Resumo: deve ser possível uma classe estender outra, e sobrescrever os métodos com sua própria implementação.

9. **Descreva: Interface Segregation Principle (Princípio da Segregação da Interface)**
   Uma classe não deve ser forçada a implementar interfaces e métodos que não irão utilizar.

10. **Descreva: Dependency Inversion Principle (Princípio da inversão da dependência)**
    Alto nível de abstração e baixo nível de acoplamento.

11. **O que é CQRS?**
    CQRS significa Command Query Responsibility Segregation. Como o nome já diz, é sobre separar a responsabilidade de escrita e leitura de seus dados. CQRS é um pattern, um padrão arquitetural assim como Event Sourcing, Transaction Script e etc. O CQRS não é um estilo arquitetural como desenvolvimento em camadas, modelo client-server, REST e etc.

12. **Usando CQRS devemos segregar as responsabilidades da aplicação em duas partes, quais são elas?**
    Command – Operações que modificam o estado dos dados na aplicação.
    Query – Operações que recuperam informações dos dados na aplicação.

13. **CQRS é arquitetura?**
    Não é! Conforme foi abordado o CQRS é um pattern arquitetural e pode ser implementado em uma parte específica da sua aplicação para um determinado conjunto de dados apenas.

14. **O que é DDD?**
    O Domain Driven Design combina práticas de design e desenvolvimento. Oferece ferramentas de modelagem estratégica e tática para entregar um software de alta qualidade. O objetivo é acelerar o desenvolvimento de software que lidam com complexos processos de negócio.
    Em seus princípios, DDD é sobre discussão, escuta e compreensão. Todo um esforço para centralizar o conhecimento.

15. **O que não é DDD?**
    O DDD não é uma tecnologia ou uma metodologia. Pode ser utilizado independente da linguagem. Não importa se é C# ou Java, se é MVC ou Windows Forms.
    Não é arquitetura em camadas e não impõe processos rígidos ao time.

16. **Porque utilizar DDD**
    Alta qualidade no software, acelera o desenvolvimento de software que lidam com complexos processos de negócio. Resulta em benefícios a longo prazo. Agiliza os processos da empresa. Facilita a implementação de novas features. Aumenta a vida útil do software.

17. **O que é Repository Pattern?**
    É um padrão de projeto que isola a camada de acesso a dados (DAL) com a camada de negócio, mais conhecida como camada de domínio.
    Permite um encapsulamento da lógica de acesso a dados, impulsionando o uso da injeção de dependência (DI) e proporcionando uma visão mais orientada a objetos das interações com a DAL.

18. **Quais são os benefícios ao utilizar Repository Pattern?**
    Permitir a troca do banco de dados utilizado sem afetar o sistema como um todo.
    Código centralizado em um único ponto, evitando duplicidade.
    Facilita a implementação de testes unitários.
    Diminui o acoplamento entre classes.
    Padronização de códigos e serviços.