**Princípios em Projeto de Software**

**Atividades de Aprendizado e Avaliação**

**Aluno: João Pedro Rodrigues Leite RA: a2487055**

**Data: 21/04/2024**

**Use esta cor no seu texto**

1. Considerando o texto no link “Inversão de Controle & Injeção de Dependência”,
   1. A inversão de controle pode ser entendida como a mudança do conhecimento que uma classe tem em relação à outra.
   2. Na primeira versão da classe VendaDeProduto, o problema é o acoplamento forte que essa classe tem em relação à classe Log.
   3. Abrir o código fonte da classe VendaDeProduto para mudar o nome do arquivo de log? Comente (3 a 5 linhas)

R: Não é o adequado a se fazer, pois o simples fato de uma mudança no nome do arquivo faria com que fosse necessário alterar o nome do arquivo na classe VendaDeProduto, mas ela não tem nada a ver com isso. Ainda que alterar o nome do arquivo em apenas uma classe é fácil, mas se tivéssemos 30 classes? Seria necessário alterar o nome do arquivo em cada uma delas.

* 1. O que a classe VendaDeProduto sabe sobre a classe Log? Comente (2 a 3 linhas)

R: Sabe criar a classe Log, e pior ainda, sabe que a classe log precisa de um nome, e até o nome do arquivo. Ela sabe demais.

* 1. A inversão de controle se dá pela mudança na estrutura do código, de modo que as responsabilidades passam a ser invertidas Assim, a classe VendaDeProduto não mais necessita de conhecimento sobre a instanciação da classe Log.
  2. No padrão “*Constructor Injection*” as dependências são injetadas via construtor.
  3. A injeção de dependência torna possível e simples a escrita e execução de testes automatizados.
  4. A inserção de uma interface definindo os serviços da classe Log reduziria ainda mais o acoplamento

1. Considerando o conteúdo do vídeo “SOLID fica FÁCIL com Essas Ilustrações”
   1. Porque o ROBO MULTIFUNCIONAL quebra o princípio “S” do SOLID? Comente (2 a 3 linhas)

R: Pois o princípio Single Responsability diz que uma classe, componente, método deve ter apenas uma responsabilidade, e o Robô multifuncional faz várias coisas ao mesmo tempo e isso está ferindo o primeiro princípio do SOLID

* 1. Com unidades independentes e isoladas você consegue
     1. Reaproveitamento de código
     2. Refatorar
     3. Testes automatizados
     4. Menos bugs, e mesmo que gere bugs você consegue isolar e consertar mais facilmente;
  2. Em um software com alto acoplamento, basta um componente no lugar errado para manchar todo o sistema com algum mal comportamento
  3. O nome da função ou componente deve expressar tudo o que ele está fazendo
  4. O princípio Open/Closed prescreve que deve ser possível adicionar novas funcionalidades sem modificar a classe base
  5. No princípio Open/Closed, a classe deve estar aberta para extensão mas fechada para modificação.
  6. Uma forma de garantir a extensão sem quebrar o princípio Open/Closed se dá pelo conceito de Polimorfismo.
  7. Respeitar o Princípio de Liskov força fazer abstrações no nível certo e ser mais consistente
  8. O exemplo do “pinguim” demonstra a quebra do Princípio da Substituição de Liskov. A abstração “Ave” não está adequada, pois nem toda ave voa
  9. O Princípio da Interface segregation promove a especificação de interfaces coesas e mais específicas
  10. O Princípio da Injeção de Dependências defende que uma classe/módulo não deve depender de outra classe/módulo, mas sim dos contratos que este último oferece.
  11. No contexto do *Dependency Injection Principle* a classe depende dos serviços definidos em uma interface, ou seja, ela não possui acoplamento com a classe que faz a implementação dos serviços, desconhecendo sua existência.
  12. Os princípios SOLID foram especificados em 1996 por Robert C. Martin.