Utilização dos 5 princípios SOLID para melhoria da qualidade de código-fonte

Gabriel Lopes Costa*

Fabrícia Pires Souza Leal**

RESUMO

Apesar da reputação que os princípios SOLID vem recebendo nos últimos anos, é essencial entender se auxiliam para a maior longevidade dos software e a importância de aplicá-los, e se possibilitam uma manutenção mais assertiva e menos custosa. Para localizar os benefícios dos princípios foram implementados exemplos aproximados da utilização no dia-a-dia, explicando os benefícios e malefícios que poderiam surgir da aplicação e correlacionando com características de manutenibilidade definidas pela ISO/IEC 25010. Com a análise das implementações pode-se concluir que a implementação do SOLID traz diversas contribuições para a qualidade de software ao ajudar os desenvolvedores a modelar as classes corretamente.

Palavras-chave: SOLID. Engenharia de Software. Qualidade de Código-fonte. Programação Orientada a Objetos.

ABSTRACT

Despite the reputation that the SOLID principles have received in last years, it is essential to understand if they help for the greater longevity of the software and the importance of applying them, and if they allow a more assertive and less costly maintenance. In order to demonstrate the benefits of the principles, approximated examples of their daily use were implemented, explaining the benefits and harms that could arise from the application and correlating with maintainability characteristics defined by ISO/IEC 25010. Based on analysis of the implementations, it can be concluded that the implementation of SOLID brings several contributions to software quality by helping developers to model classes correctly.

Keywords: SOLID. Software Engineering. Source Code Quality. Oriented Object Programming.

---

* Rede de Ensino Doctum – Unidade Caratinga – devgabrielcosta@gmail.com – Graduando em Ciência da Computação

** Rede de Ensino Doctum – Unidade Caratinga – fabricia@doctum.edu.br – Mestre em Informática – Orientadora do trabalho

1 - Introdução

A POO (Programação orientada a objetos) surgiu da evolução no que se conhece hoje como engenharia de software na década de 60, termo difundido com a crise do software que aconteceu devido a complexidade na resolução dos problemas, má qualidade das linguagens de programação, além do estouro de orçamentos, cronogramas e custos dos projetos de softwares na época. O objetivo era tornar mais intuitiva a leitura e baratear os custos de manutenção do software.

Mesmo com o surgimento da POO, ainda era necessário especificar formas de se obter as vantagens que se tinham como objetivo com a utilização das linguagens baseadas no paradigma. A orientação a objetos quando utilizada de forma incorreta e desorganizada pode gerar diversos problemas, e não passa de código estruturado organizado de forma diferente.

O envelhecimento de um software costuma ser um problema, seja pelas tecnologias adotadas, pela falta de testes, ou falhas que começam desde a arquitetura e do design do sistema até a inserção de novas falhas em manutenções posteriores. Falhas de design são mais perceptíveis quando se torna difícil fazer a manutenção ou criação de novas funcionalidades, uma dessas falhas sendo a má gestão das dependências (MARTIN, 2005).

Com anos da criação da programação orientada a objetos, os 5 princípios de design conhecidos pelo acrônimo SOLID, surgiram com o objetivo de especificar como obter tais vantagens através da gestão das dependências e está há alguns anos na indústria, tendo cada vez mais a atenção em vagas de grandes empresas e implementação em projetos mais novos. Segundo Robert C. Martin (2005) os princípios com a gestão correta das dependências resolve problemas relacionados ao desenvolvimento de software, como a alta complexidade, baixa eficiência, baixa reutilização, e grande quantidade de retrabalho, tornando o código fácil de ser modificado, robusto e reutilizável.

Mesmo anos após a criação ainda não se tem a devida atenção em como o SOLID ajuda a resolver as falhas de design que podem ocorrer e como ajuda na longevidade do software. Sendo assim, foi implementada e analisada a eficiência da utilização de cada princípio de forma isolada, no cenário que propõem a melhorar, aproximando os exemplos criados ao máximo da utilização no dia a dia,

com o objetivo de entender principalmente a eficácia e justificar a utilização para melhoria da qualidade de código.

Para os testes, em cada um dos 5 princípios foram adotadas situações em que é possível perceber a mudança que a implementação traz. Os exemplos são representações de situações que podem ocorrer no dia a dia como a comunicação da aplicação com o banco de dados, a implementação de uma regra de negócio, modelagem de classes e de interfaces. Após os exemplos de código foi apresentado uma explicação dos pontos que o princípio testado melhora da perspectiva da ISO/IEC 25010, e uma descrição de como a solução melhora o problema.

2 - Referencial Teórico

2.1 Engenharia de software: Desenvolvimento de software

Segundo Pressman (2016, p. 4) software pode ser definido como:

> […] (1) instruções (programas de computador) que, quando executadas, fornecem características, funções e desempenho desejados; (2) estruturas de dados que possibilitam aos programas manipular informações adequadamente; e (3) informação descritiva, tanto na forma impressa quanto na virtual, descrevendo a operação e o uso dos programas.

Pode-se entender o desenvolvimento de software como o processo desde as escolhas de tecnologia, a tradução dos requisitos para tarefas e a própria programação. A programação consiste em escrever tais instruções em uma determinada tecnologia, a fim de realizar alguma ação desejada com as informações que se possui. Todo o processo de organização desse desenvolvimento, desde a escolha do melhor design e metodologia, pode ser chamado de engenharia de software.

Muitos problemas podem causar dificuldades na longevidade de um software, seja no próprio funcionamento ou no custo para mantê-lo. Há diversos autores que pontuam esses problemas, e qual caminho tomar para evitá-los.

Relacionado principalmente ao código-fonte pode-se pontuar problemas de design, como acoplamento e duplicidade, complexidade, código sem testes e/ou difícil de ser testado (THOMAS, HUNT, 2019).

Existem metodologias como *Don’t Repeat Yourself* (Não repita a si mesmo) e

Keep it simple, stupid (mantenha isso simples, estupido), para orientar sobre duplicidade e complexidade respectivamente. Diversos padrões de design tem como objetivo tornar a estruturação do código mais previsível, não deixando puramente a cargo da interpretação do programador, simplificando a manutenção ou a implementação de mais funcionalidades posteriormente por outras pessoas. O SOLID foi escrito numa tentativa de listar alguns princípios para o desenvolvimento da programação orientada a objetos, ajudando a extrair todos os benefícios que o surgimento deste paradigma de programação proporcionou.

2.1.1 Paradigma Imperativo: Orientação a objetos

O paradigma imperativo é um dos principais paradigmas de programação, e é o responsável pela forma que se desenvolve software nos dias atuais. Especificamente a POO surgiu em meados dos anos 60 e foi amplamente difundida na década de 90 com a popularização da linguagem de programação C++. Sobre a Orientação a objetos pode-se afirmar:

> [...] A tecnologia de objetos é um esquema de empacotamento que facilita a criação de unidades de software significativas. Essas unidades são grandes e focalizadas em áreas de aplicação específicas[...] Os objetos têm propriedades (isto é, atributos, como cor, tamanho e peso) e executam ações (isto é, comportamentos, como mover, dormir ou desenhar). Classes representam grupos de objetos relacionados [...] (Deitel et al, p. 54)

Com a tecnologia de objetos, o foco é aproximar o código-fonte do mundo real, ao ser possível adicionar atributos e ações torna-se possível representar qualquer cenário por meio do código.

Além da dificuldade de manter e dos problemas que traz em grandes softwares, a programação estruturada falha em espelhar o mundo real e acaba sendo complexa de entender, (Deitel et al, p. 54) principalmente por se basear em ações invés de substantivos. Apesar desta base não ser a principal melhoria, a POO segundo algumas organizações, tende a produzir software mais organizado e com menos requisitos de manutenção, sendo que 80% dos custos de desenvolvimento estão relacionados à manutenção e não ao desenvolvimento inicial. (Deitel et al, p. 54).

Na orientação a objetos, tem-se funcionalidades que representam o paradigma, podendo serem entendidas segundo Bruce Eckel (2006, p. 145 e seg.)

como:

- Classe: Representa um conjunto de objetos, com suas propriedades e ações.
- Objeto: Instância de uma classe, representa o estado de um objeto específico.
- Encapsulamento: Torna a execução de um trecho de código o mais isolado possível. Através de palavras reservadas da linguagem limita o acesso aos métodos e classes.
- Herança: Permite às classes estenderem outras classes e compartilharem suas ações e atributos.
- Polimorfismo: Possibilita um mesmo atributo ou método ser implementado de forma diferente em objetos que compartilham a mesma classe base.

  Esses recursos são utilizados ao criar e estender classes, interfaces e métodos.

A POO diz respeito ao modo que o software funciona e os recursos adicionados, mas que por si só não garantem as melhorias propostas pelo paradigma, pois podem ser utilizados de maneira incorreta.

Metodologias, padrões, surgiram após a criação da orientação a objetos para reforçar suas qualidades, auxiliar no software ágil, nas manutenções ao longo dos anos. Métricas como coesão e acoplamento são utilizadas desde a programação estruturada e foram transportadas para a POO, e o próprio SOLID utiliza do conceito para basear suas contribuições.

2.1.2 Coesão e Acoplamento

Coesão e acoplamento são conceitos qualitativos para avaliar a independência entre as classes do software. “Um módulo coeso realiza uma única tarefa, exigindo pouca interação com outros componentes em outras partes de um programa. De forma simples, um módulo coeso deve (de maneira ideal) fazer apenas uma coisa.“ (PRESSMAN, 2016, p. 236). Portanto a alta coesão é a separação de funcionalidades de forma a tornar mais específica a responsabilidade da classe. “O acoplamento é uma indicação da interconexão entre os módulos em uma estrutura de software. Ele depende da complexidade da interface entre os módulos, do ponto onde é feito o acesso a um módulo e dos dados que passam pela interface” (PRESSMAN, 2016, p. 236). Entendendo assim que quanto menor o

número de classes externas que forem utilizadas, menor o acoplamento.

Figura 01: Coesão e Acoplamento

Fonte: De autoria própria.

Na figura 01 pode-se observar como a comunicação acontece no cenário ideal e no que se deve evitar. Com a alta coesão e o baixo acoplamento a comunicação é realizada entre poucas unidades de código, demonstrando a responsabilidade bem definida entre as classes e a troca de mensagens em menos locais.

As duas medidas estão diretamente relacionadas. Ao separar responsabilidades aumenta-se a coesão da classe, mas resulta na necessidade de utilizar classes externas, aumentando o acoplamento, e ao juntar várias responsabilidades em uma mesma classe, tem-se o baixo acoplamento, mas diminui a coesão pela classe ter mais de uma responsabilidade.

O ideal seria ter um código com alta coesão e nenhum acoplamento, mas devido a comunicação entre as classes serem necessárias esse cenário acaba não sendo possível a depender do contexto. O principal problema ao se ter o acoplamento seria propagar as mudanças em classes por todo o software, sendo remediado com a utilização do SOLID.

2.2. SOLID

Os 5 princípios de design da POO foram definidos por Robert C. Martin por

volta do ano de 2000, e visam parametrizar uma maneira de aproveitar ao máximo os benefícios do paradigma. O nome foi definido por Michael Feathers após notar que as iniciais de cada princípio poderiam se encaixar no acrônimo *Single Open Liskov Interface Dependency* (SOLID), sendo os princípios conhecidos como *Single Responsability Principle (*SRP), *Open-Closed Principle* (OCP), *Liskov Substitution Principle* (LSP), *Interface Segregation Principle*(ISP), *Dependency Inversion Principle*(DIP).

Segundo Martin, estes 5 princípios expõem os aspectos de gestão de dependência da POO em oposição aos aspectos de conceituação e modelagem. Sobre a gestão de dependência pode-se afirmar:

> A gestão de dependência é um problema que a maioria de nós enfrenta. Sempre que trazemos em nossas telas um monte de código legado emaranhado, estamos experimentando os resultados de um mau gerenciamento de dependência. O mau gerenciamento de dependência leva a um código difícil de mudar, frágil e não reutilizável. (MARTIN, 2005)

Com isso, ao resolver a gestão de dependências o objetivo dos princípios é sobretudo tornar o código flexível, robusto e reutilizável, mantendo assim um software de qualidade ao longo das manutenções e conseguindo os benefícios visados ao utilizar a programação orientada a objetos.

O princípio da responsabilidade única (SRP) afirma que cada módulo de software deve ter uma única razão para mudar (MARTIN, 2014) e está diretamente relacionado com o conceito de coesão. Cada função deve ser responsável por executar apenas uma ação e cada classe um conjunto de ações do mesmo contexto. Trechos grandes de código são difíceis de manter e acabam sendo alterados frequentemente, quando divididos em partes menores, deixam de propagar os erros por consequência em locais que não estão relacionados diretamente aquela tarefa e ficam mais fáceis de serem localizados e corrigidos. Para seguir o princípio basta que a classe não esteja realizando ações que não são de sua responsabilidade, por exemplo em uma rota de uma aplicação qualquer é necessário validar os dados, realizar as regras de negócio e salvar no banco de dados, em algum lugar esse código precisa ser chamado e executado, mas não deve ser definido em uma única classe e método, ao criar classes para validação, para comunicação com o banco de dados, e instanciá-las na classe principal já passa-se a seguir o princípio.

O princípio aberto/fechado (OCP) orienta a escrever as classes de forma que elas podem ser estendidas sem precisarem ser modificadas (MARTIN, 2000, p. 4). Isso é, escrever classes que são abertas para extensão mas fechadas para modificação. Ser fechada para modificação significa modificar a classe de forma reduzida já que alterar constantemente uma classe pode indicar que ela foi modelada de forma incorreta. Em uma classe que executa determinada regra de negócio significa que ao precisar estender essa regra não seja necessário modificar a classe que executa a regra diretamente, apenas criar novas classes que podem ser passadas a ela e usufruir da sua ação.

No princípio de substituição de Liskov (LSP) é definido que classes derivadas devem ser substituíveis por suas classes bases (MARTIN, 2000, p. 8). Está relacionado com uma das principais justificativas da POO, a reutilização de código, que quando realizado de maneira incorreta traz diversos erros para a aplicação. É comum acontecer erros ao estender classes, alterando contratos pré-estabelecidos, isso pode ser corrigido ao realizar o design da aplicação, pensando realmente se as classes estendidas são correlacionadas, e se os métodos da classe base serão utilizados pela classe herdeira. É comum fazer extensões para se aproveitar código e atributos que apesar de se terem os mesmos nomes e tipos não são iguais.

No princípio de segregação de interfaces (ISP) se diversas outras classes implementam a interface, em vez de fazer uma interface que supre todos os clientes, deve-se desmembrar em interfaces menores e específicas (MARTIN, 2000, p. 14). O objetivo com uma interface específica entre a comunicação, é evitar modificar a interface futuramente arriscando sem necessidade quebrar os contratos das classes clientes. Quando um cliente da interface genérica não implementar um dos métodos, ele teria de implementar um método inútil e retornar vazio ou uma exceção, quando desmembrado em interfaces menores o cliente pode estender apenas as interfaces necessárias. Interfaces não devem ser genéricas, devem especificar quais comportamentos determinada classe deve seguir para implementar o contrato, implementar um método retornando uma exceção para reaproveitar uma interface existente é também uma quebra de contrato pois nunca executaria a ação esperada.

O Princípio da inversão de dependências (DIP) é sobre depender de interfaces, funções e classes abstratas ao invés de funções e classes concretas

(MARTIN, 2000, p. 12). Propõe-se a resolver principalmente os problemas causados pelo acoplamento, na qual alterações a determinado código torna necessário retrabalho nas classes principais por propagar problemas. É solucionado ao depender de classes abstratas e interfaces, não importando qual objeto será enviado para realizar as ações correspondentes, basta que esteja seguindo o contrato estabelecido, ao modificar a interface ou quebrar contrato em alguma classe que o implementa, os erros são notificados e a necessidade estaria explícita.

Diante do conhecimento de cada um dos princípios, para analisar a qualidade da abordagem proposta pelo SOLID se faz necessário medir os benefícios que a implementação traz. Para definição das métricas de qualidade do software, pode ser utilizada a norma ISO/IEC 25010.

2.3. ISO/IEC 25010

A norma ISO/IEC 25010 (2011), é um padrão para controle de qualidade de software, que define um conjunto de parâmetros com o objetivo de padronizar as avaliações de um software e surgiu da evolução da norma ISO/IEC 9126 (1991). O modelo determina quais características devem ser levadas em conta ao avaliar a qualidade de um software, sendo essas características: Adequação funcional, performance, compatibilidade, usabilidade, integridade, segurança, manutenção e portabilidade. E cada uma delas dividida em diversas subcaracterísticas. Na Figura 02 segue as características e subcaracterísticas de cada parâmetro da ISO/IEC 25010:

Figura 02: Quadro de características

Fonte: iso25000.com

Ainda segundo a ISO/IEC 25010 pode-se entender as subcaracterísticas de manutenibilidade como:

- Modularidade: Grau em que um sistema ou programa de computador é composto de componentes discretos, de modo que uma mudança em um componente tenha impacto mínimo em outros componentes.
- Reusabilidade: Grau em que um ativo pode ser usado em mais de um sistema ou na construção de outros ativos.
- Analisabilidade: Grau de eficácia e eficiência com que é possível avaliar o impacto em um produto ou sistema de uma mudança pretendida em uma ou mais de suas partes, ou diagnosticar um produto quanto a deficiências ou causas de falhas, ou identificar peças a serem modificadas.
- Modificabilidade: Grau em que um produto ou sistema pode ser modificado de forma eficaz e eficiente sem introduzir defeitos ou degradar a qualidade do produto existente.
- Testabilidade: Grau de eficácia e eficiência com que os critérios de teste podem ser estabelecidos para um sistema, produto ou componente e os testes podem ser realizados para determinar se esses critérios foram atendidos.

Desta forma a ISO apresenta critérios qualitativos para melhorar e atingir um nível de qualidade maior no software.

A Qualidade do código-fonte está diretamente relacionado ao aspecto de manutenibilidade, para ter um código de boa qualidade deve se cumprir os objetivos levantados pelas subcaracterísticas, e por consequência ajuda a alcançar mais facilmente outras características, pois diversas dessas estão relacionadas diretamente com a capacidade do código evoluir.

Com o entendimento do que é POO e o seu objetivo, o surgimento do SOLID e tendo como base uma forma de avaliação, é possível avaliar o desempenho de sua utilização e entender como esses benefícios podem auxiliar na melhoria dos softwares produzidos.

Ao avaliar a aplicação dos princípios o objetivo é principalmente tornar claro o foco de sua implementação, demonstrando como pode impactar positivamente a evolução do software, reduzindo o esforço e facilitando o entendimento do código.

3 - Metodologia

Buscando evidenciar os benefícios da utilização do padrão SOLID, visando

pontuar detalhadamente a contribuição para a qualidade de código, foi realizada uma pesquisa aplicada, de caráter exploratório.

O próprio SOLID não se restringe a linguagem, ou algum cenário específico e pode ser utilizado em qualquer aplicação. Apesar da linguagem de programação escolhida, os benefícios pontuados são independentes de linguagem desde que a mesma implemente a orientação a objetos. Portanto os benefícios descritos no trabalho podem ser obtidos em qualquer aplicação.

O levantamento de dados foi feito com a implementação de exemplos na linguagem C# com trechos de código violando cada um dos princípios de design e posteriormente a implementação da solução para cada uma das violações.

Os arquivos foram organizados em projetos isolados para cada princípio, contendo os arquivos em comum entre a solução e violação na pasta raiz e os arquivos que foram alterados em cada uma das implementações na suas respectivas pastas nomeadas como solução e violação.

Os projetos contêm apenas o código necessário para realizar a comparação e demonstrar o que exatamente está mudando, sem métodos, atributos e código excessivos. Para ser realizada uma análise individual de cada princípio, a correção foi realizada apenas em relação ao princípio a ser analisado no projeto, nunca levando mais de um princípio em consideração por vez.

Os exemplos foram aproximados de situações reais, que ocorrem durante a produção de software como encadeamento de *if's*, utilização de implementações ao invés de abstrações, agrupamento de código em uma mesma classe, cenários que podem ser observados em aplicações web e desktop. Foram escolhidos exemplos relacionados a cadastro de produtos, cadastro de pessoas, regras para compra de cursos por alunos e sistema para gerenciamento de funcionários.

Quanto ao algoritmo de cada projeto, para o princípio da responsabilidade única foi elaborada uma classe Service responsável por centralizar as ações de uma classe Pessoa, nesta Service foi implementado um método único realizando todas as subações necessárias para executar a ação escolhida e depois foi realizada a separação em classes diferentes para cada ação de forma coerente.

No princípio aberto/fechado foi feita uma classe de cálculo responsável por realizar o cálculo de um desconto baseado nos alunos que foram recebidos, posteriormente a correção irá reescrever a solução de forma que a classe não

precise ser alterada para realizar o cálculo em novos alunos implementados posteriormente.

No princípio de substituição de Liskov foi realizada uma herança, de forma que a classe filha herda métodos e atributos desnecessários da classe pai, para a correção a herança foi desfeita e as classes separadas corretamente, corrigindo problemas que poderiam surgir com a implementação de novas funcionalidades posteriormente.

Para o princípio de segregação de interfaces foi criado uma interface genérica responsável por definir as assinaturas do repositório de produto, e foi realizada a implementação incorreta por um segundo repositório. Na correção a interface foi dividida em duas interfaces específicas para garantir que todos os métodos necessários sejam implementados pelo repositório cliente.

Para o princípio da inversão de dependência foi criado um controller que instancia suas dependências internamente no construtor, na correção as dependências foram injetadas através da interface. A interface é utilizada para garantir que a classe injetada cumpra o contrato necessário, possibilitando ao controller lidar com diferentes classes desde que implementem a interface, tornando explícita a necessidade de tais métodos, e permitindo a criação de testes mais assertivos.

Devido os benefícios apresentarem melhorias como redução de linhas duplicadas mas não ter base em critérios quantitativos, foi realizada a comparação da violação e solução e gerado um estudo qualitativo relacionado especialmente com a ISO/IEC 25010, descrevendo os possíveis benefícios dos princípios, e se ajudam nas subcaracterísticas de manutenibilidade.

Para demonstração dos resultados cada um dos princípios foram separados em subtítulos, contendo uma breve descrição do problema, imagens demonstrando o código fonte utilizado na violação e posteriormente na solução, descrição da violação e descrição da solução que foi utilizada para seguir o princípio. As imagens contém apenas o código principal, podendo os arquivos auxiliares serem consultados nos apêndices do artigo. Após a contextualização da violação e solução proposta foi descrito os pontos observados, quais benefícios podem surgir da utilização e o que pode desencadear no projeto.

4 - Resultados

4.1. SRP - Princípio da responsabilidade única

Para o princípio da responsabilidade única a violação foi realizada em uma classe de serviço responsável por controlar as regras de negócio da inclusão de uma pessoa na base de dados.

Figura 03: Classe PessoaService - Violação

```
5   public class PessoaService
6   {
7       private readonly SqlConnection _sqlconn;
8       private const string _connectionString = "DATABASECONNECTIONSTRING";
9
        0 referências
10      public PessoaService()
11      {
12          _sqlconn = new SqlConnection(_connectionString);
13      }
14
        0 referências
15      public Pessoa? InserirPessoa(Pessoa pessoa)
16      {
17          if (String.IsNullOrEmpty(pessoa.Nome))
18              return null;
19          if (ValidarCPF(pessoa.CPF))
20              return null;
21          if (pessoa.DataNascimento.AddYears(-18) > DateTime.Today.AddYears(-18))
22              return null;
23
24          _sqlconn.Open();
25
26          var sqlCommand = new SqlCommand(
27              $"INSERT INTO pessoa " +
28              "(Nome, Email, DataNascimento, CPF)" +
29              "VALUES" +
30              "({pessoa.Nome}, {pessoa.Email}, {pessoa.DataNascimento}, {pessoa.CPF})"
31          );
32
33          var sqlData = sqlCommand.ExecuteReader();
34
35          _sqlconn.Close();
36
37          if (sqlData.RecordsAffected > 0)
38          {
39              while (sqlData.Read())
40              {
41                  pessoa.Id = (int)sqlData["Id"];
42              }
43
44              return pessoa;
45          }
46
47          return null;
48      }
```

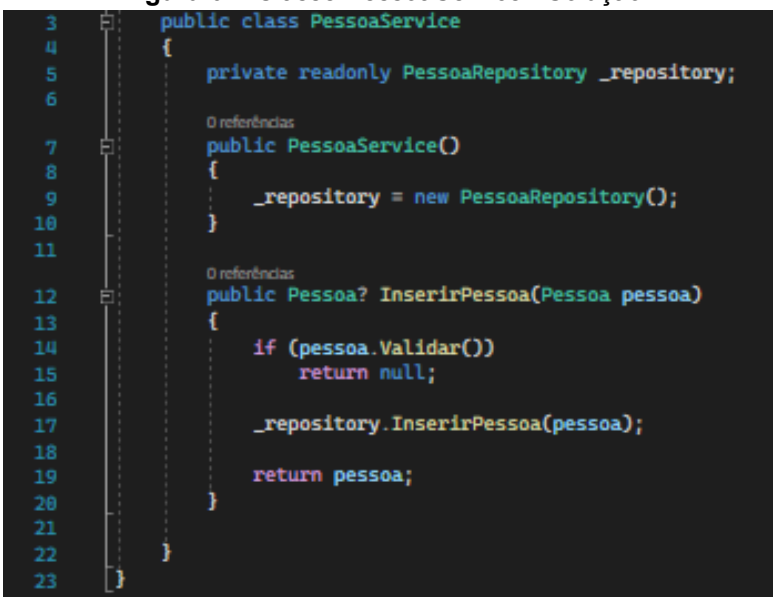
Fonte: De autoria própria.

Conforme a implementação da Figura 03 (linha 15 a 48), o foco da análise é no método InserirPessoa, o método recebe a classe Pessoa (linha 15), realiza a

validação de seus atributos (linha 17 a 22), após a validação abre conexão com o banco de dados (linha 24), define e executa o comando (linha 26 a 33) e inclui o id no model para retorno (linha 37 a 45).

Na classe há um acúmulo de responsabilidades, o Service é responsável por centralizar a lógica mas não deveria ser responsável por definir cada ação. Ao alterar uma dessas ações poderia causar falhas de comportamento nas demais, e nenhum dos comportamentos podem ser reaproveitados em outras partes do software, se necessário criar um registro ou validar os dados da classe Pessoa em outro local do software, seria necessário duplicar o código.

Figura 04: Classe PessoaService - Solução



```
3   public class PessoaService
4   {
5       private readonly PessoaRepository _repository;
6
        0 referências
7       public PessoaService()
8       {
9           _repository = new PessoaRepository();
10      }
11
        0 referências
12      public Pessoa? InserirPessoa(Pessoa pessoa)
13      {
14          if (pessoa.Validar())
15              return null;
16
17          _repository.InserirPessoa(pessoa);
18
19          return pessoa;
20      }
21
22  }
23  }
```

Fonte: De autoria própria.

Ao aplicar o princípio conforme a Figura 04 (linha 12 a 20) o método InserirPessoa passa a apenas centralizar as regras e executar as ações, mas a implementação está sendo realizada em classes divididas por responsabilidade. Como observado na solução proposta (apêndice A, 2), para correção foi criada uma nova classe CPF, a validação dos atributos passa a ser realizada na própria entidade Pessoa e para conexão e inserção no banco de dados foi criado uma classe de repositório responsável pelas consultas da classe Pessoa.

O SRP facilita a modificação e a analisabilidade por separar em arquivos específicos e tornar mais claro o objetivo de cada classe e método, sendo possível alterar por exemplo somente os atributos e métodos da classe Pessoa e sua validação e propagar essas mudanças para todas as classes onde estiver sendo

utilizada. O SRP torna possível também a reusabilidade de métodos, auxiliando a utilizar em mais trechos do sistema, ou em outros sistemas a depender da implementação, diminuindo a redundância de ações, como o tipo CPF e sua validação que pode ser utilizada em qualquer classe do software. Apesar de não ser possível controlar totalmente o comportamento interno das dependências com a aplicação do SRP a aplicação de testes passa a analisar unidades menores de código podendo testar comportamentos específicos mais facilmente.

O SRP auxilia em todas as subcaracterísticas de manutenção da ISO/IEC 25010 ou a alcançá-las mas é o responsável por tornar o código coeso, e gerar alto acoplamento. Apesar de o código estar definido em lugares diferentes, as classes principais ainda precisam ser acessadas de alguma forma aumentando a comunicação entre as diferentes classes, portanto ao realizar mudanças nessas classes como a alteração do nome do método, parâmetros de entrada, tipo do retorno ou a ação a ser realizada, essas alterações podem ser propagadas para as classes que utilizam o método, fazendo com que as respectivas classes deixem de funcionar corretamente, para prevenir tais erros torna-se necessário a utilização de interfaces.

4.2. OCP - Princípio aberto/fechado

No princípio aberto/fechado foi implementado uma calculadora para a regra de negócio. A CalculadoraPrecos é responsável por retornar o valor de um curso com um desconto de acordo com o tipo de aluno que está o adquirindo.

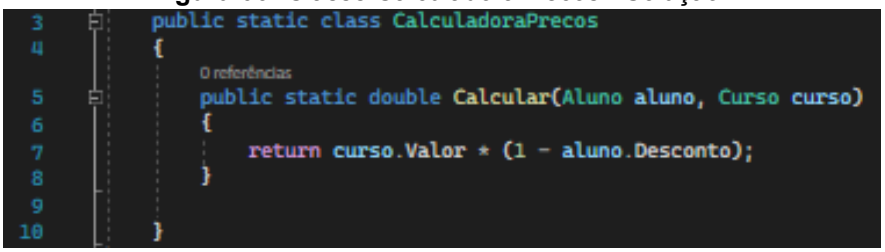
Figura 05: Classe CalculadoraPrecos - Violação

```
 9    public static class CalculadoraPrecos
10    {
11
          0 referências
12        public static double Calcular(Aluno aluno, Curso curso)
13        {
14            if (aluno.Tipo.Equals(TipoAluno.TipoA))
15                return curso.Valor * 0.9;
16            else if (aluno.Tipo.Equals(TipoAluno.TipoB))
17                return curso.Valor * 0.7;
18
19            return curso.Valor * 0.7;
20        }
21
22    }
```

Fonte: De autoria própria.

Conforme a figura 05 no método Calcular recebe-se as classes Aluno e Curso, e baseado nos tipos de alunos é concedido desconto no valor total do curso, o aluno TipoA tem 10% de desconto e o aluno TipoB tem 30% de desconto. A classe não tem problemas aparentes, mas a todo momento que for necessário incluir um novo tipo de aluno ou alterar o valor do desconto será preciso alterar um método que deveria apenas calcular. Se a forma de calcular não mudou, a classe não deveria precisar mudar.

Figura 06: Classe CalculadoraPrecos - Solução



Fonte: De autoria própria.

Na solução apresentada na figura 06, o método Calcular passa a utilizar informações que serão retiradas diretamente da classe Aluno para realizar o cálculo, mas neste caso o sistema não tem apenas uma implementação de aluno.

Como demonstrado (apêndice B, 2) foi modificada a classe Aluno para ser abstrata e os diferentes tipos de Alunos devem ser implementados e herdar da classe Aluno. Desta forma, para estender o funcionamento da classe Calculadora, apenas precisa-se definir novos tipos de Aluno no código com seus respectivos percentuais e a classe já passa a calcular o desconto para o novo tipo.

No OCP se contribui principalmente para as subcaracterísticas de analisabilidade e modificabilidade. Acaba a necessidade de existir uma cadeia de *if's* enorme, no exemplo demonstrado existe apenas uma decisão para tomar, mas se houvesse descontos diferentes para pares de aluno e cursos diferentes, a complexidade do método poderia aumentar significativamente. O OCP auxilia também na modificabilidade pois para ser implementado um novo tipo de aluno ou modificar a regra para um criado anteriormente, não é necessário a alteração na classe responsável pelo cálculo.

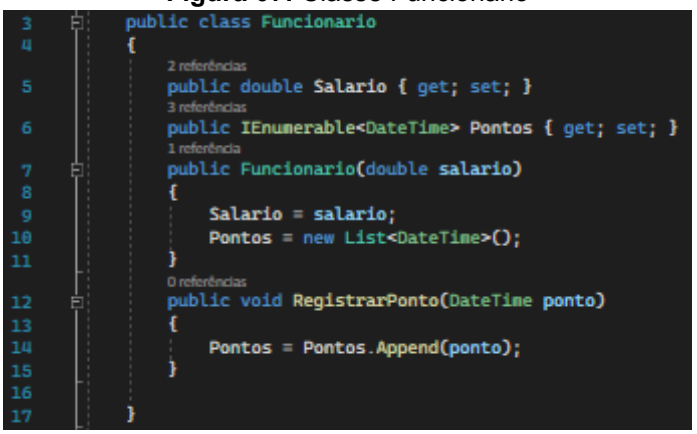
A implementação do OCP em uma classe por si só não gera desvantagens, mas a forma a ser utilizada para tornar a classe aberta a modificações deve ser observada, no exemplo apresentado as classes que herdam de Aluno apenas tem o

objetivo de definir valores de desconto novos para cada tipo, mas caso em um momento futuro do software os tipos de alunos diferentes passassem a ter também comportamentos variados, poderiam ser implementados de forma incorreta na classe base e propagando assim para locais que não deveriam ser utilizados.

4.3. LSP - Princípio de substituição de Liskov

Para o princípio de substituição de Liskov foi elaborada uma classe referente a um funcionário de uma empresa e simulado a herança incorreta realizada por uma classe referente a um Gerente.

Figura 07: Classe Funcionario



Fonte: De autoria própria.

Conforme a Figura 07 a classe Funcionário possui os atributos Salario e Pontos, e o método de RegistrarPonto. Apesar da classe poder ter sido implementada de forma mais genérica e as demais classes mais específicas serem implementadas posteriormente, não há problemas na implementação, pois o objetivo desta aplicação é que exista diferenças entre funcionários e gerentes. O princípio trata de heranças sendo realizadas de forma incorreta, para impedir que ações funcionem onde não deveriam, ou que classes sejam enviadas a locais que não as tratem corretamente. No caso da aplicação isso ocorreria pois um gerente não deve ser tratado como um funcionário mais específico, e sim um outro tipo distinto com seus próprios tratamentos.

Figura 08: Classe Gerente - Violação

```
3   public class Gerente : Funcionario
4   {
        0 referências
5       public Gerente(double salario) : base(salario)
6       {
7       }
8
        0 referências
9       public double Bonus()
10      {
11          return Salario * 0.2;
12      }
```

Fonte: De autoria própria.

Como demonstrado na Figura 08, a herança foi feita pela classe Gerente de modo a aproveitar os atributos implementados pela classe Funcionario pois conceitualmente estão relacionados. Foi definido um método Bônus, responsável a retornar o valor da bonificação que deve ser concedida ao gerente quando for necessário.

O erro no caso demonstrado acontece ao herdar incorretamente da classe Funcionario, apesar de aparentemente estarem relacionadas, na regra de negócio implementada as duas classes devem ser tratadas de forma diferentes, uma terceira classe que trata da classe Funcionario não vai necessariamente ser capaz de tratar a classe Gerente. Mesmo o gerente tendo um salário, ele não têm a necessidade de registrar o ponto, o que faz com que a herança traga métodos que não serão utilizados.

Figura 09: Classe Gerente - Solução

```
3   public class Gerente
4   {
        2 referências
5       public double Salario { get; set; }
        0 referências
6       public Gerente(double salario)
7       {
8           Salario = salario;
9       }
10
        0 referências
11      public double Bonus()
12      {
13          return Salario * 0.2;
14      }
15
16  }
```

Fonte: De autoria própria.

Na solução aplicada na Figura 09 a classe gerente deixa de herdar da classe Funcionario e passa a implementar seus próprios atributos e métodos.

A solução apresentada do ponto de vista da ISO/IEC 25010 contribui para a subcaracterística de analisabilidade, o software torna-se mais coerente por não

reutilizar classes apenas pela questão da nomenclatura, focando em implementar a solução coerente com as modificações que a regra de negócio pode propor. Ao separar a classe Gerente as alterações na classe base não se propagam sem necessidade, e torna evidente a demanda de um tratamento específico.

Para contribuir com a reusabilidade em várias aplicações, o mesmo problema poderia ter sido resolvido com a criação de uma classe FuncionarioBase na qual tanto a classe Gerente quanto a classe Funcionario demonstradas herdam da base, e assim haveria a possibilidade de se criar mais classes genéricas que fariam o tratamento de ambas as classes quando necessário, ou de outras classes de cargo que poderiam surgir com a evolução do software.

O LSP também não apresenta um modelo pronto para ser implementado em todos os programas, ele descreve uma regra que ao ser seguida auxilia no design do software, na solução demonstrada não foi utilizado o melhor dos cenários no design mas ainda sim é possível notar onde o princípio visa melhorar a implementação.

4.4. ISP - Princípio de segregação de Interfaces

Para o princípio de segregação de interfaces foi implementada uma interface geral para representar o contrato necessário para uma classe que representa os passos necessários da compra de um produto físico.

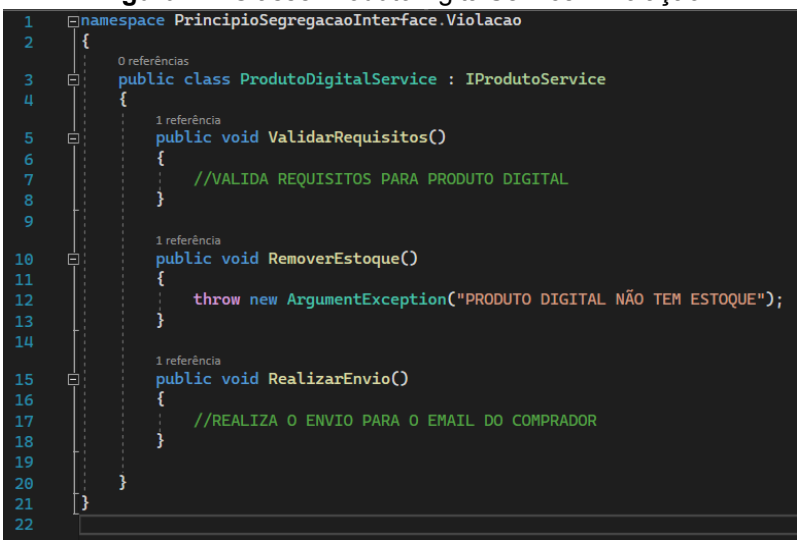
Figura 10: Interface IProdutoService - Violação

```
2 referências
public interface IProdutoService
{
    2 referências
    void ValidarRequisitos();
    2 referências
    void RemoverEstoque();
    2 referências
    void RealizarEnvio();
}
```

Fonte: De autoria própria.

A interface está correta do ponto de vista da utilização pela classe responsável pelos produtos físicos, o princípio visa chamar a atenção para quando ao invés de especificar uma nova interface para o produto digital, foi utilizada a mesma interface do produto físico.

Figura 11: Classe ProdutoDigitalService - Violação

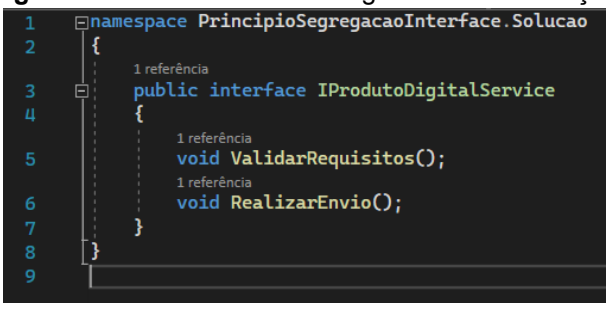


```
namespace PrincipioSegregacaoInterface.Violacao
{
    0 referências
    public class ProdutoDigitalService : IProdutoService
    {
        1 referência
        public void ValidarRequisitos()
        {
            //VALIDA REQUISITOS PARA PRODUTO DIGITAL
        }

        1 referência
        public void RemoverEstoque()
        {
            throw new ArgumentException("PRODUTO DIGITAL NÃO TEM ESTOQUE");
        }

        1 referência
        public void RealizarEnvio()
        {
            //REALIZA O ENVIO PARA O EMAIL DO COMPRADOR
        }

    }
}
```

Fonte: De autoria própria.

O ato de utilizar uma interface geral como na Figura 11, faz com que o software implemente métodos inúteis, atrapalhando assim a utilização por classes externas por não saberem exatamente quais métodos podem/devem ser utilizados. Para a correção pode-se simplesmente criar uma nova interface para especificar os métodos necessários para o produto digital.

Figura 12: Interface IProdutoDigitalService - Solução

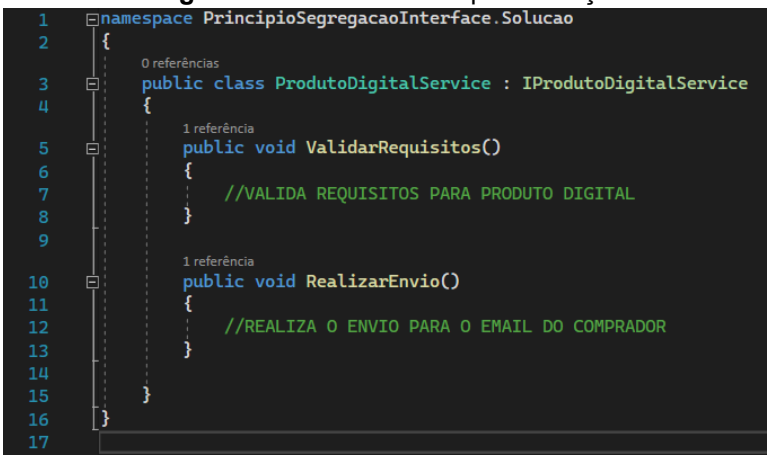


```
namespace PrincipioSegregacaoInterface.Solucao
{
    1 referência
    public interface IProdutoDigitalService
    {
        1 referência
        void ValidarRequisitos();
        1 referência
        void RealizarEnvio();
    }
}
```

Fonte: De autoria própria.

Desta forma os serviços de produto e produto digital herdam seus contratos específicos, não sendo necessário lançar exceções, e facilitando a utilização por classes que desconhecem a implementação do código.

Figura 13: Interface IEstoque - Solução



Fonte: De autoria própria.

Do ponto de vista da ISO/IEC 25010, foi contribuído principalmente para a analisabilidade por tornar o design do código mais coerente, as alterações no contrato passam a propagar apenas para classes na qual são necessárias alterações, e a utilização mais específica torna claro para classes externas quais métodos têm utilidade.

4.5. DIP - Princípio da inversão de dependências

No princípio da inversão de dependências foi simulado um controller que possui um método equivalente a uma chamada POST de uma aplicação, responsável por realizar a inserção de um produto na base de dados.

Figura 14: Classe ProdutoController - Violação

```
9   public class ProdutoController
10  {
11      private readonly ProdutoRepository _produtoRepository;
12
13
        0 referências
14      public ProdutoController()
15      {
16          _produtoRepository = new ProdutoRepository();
17      }
18
        0 referências
19      public ResultMessage<Produto> Inserir(Produto produto)
20      {
21          if (produto.Validar())
22              return new ResultMessage<Produto>(false, "Campos Invalidos", null);
23
24          _produtoRepository.Insert(produto);
25
26          return new ResultMessage<Produto>(true, "Campos Invalidos", produto);
27      }
28
29  }
```

Fonte: De autoria própria.

O foco neste princípio é na utilização de dependências externas. Conforme o construtor na linha 14 da figura 14, a única dependência do controller é o repositório, então ele realiza a instância da classe no construtor e atribui a variável. O problema não está apenas em instanciar no construtor, mas em depender de uma implementação invés de uma abstração.

Para corrigir esta violação pode-se fazer com que o repositório implemente uma interface correspondente aos seus métodos e passe a injetar esta dependência.

Figura 15: Classe ProdutoController - Solução

```
3   public class ProdutoController
4   {
5       private readonly IProdutoRepository _produtoRepository;
6
        0 referências
7       public ProdutoController(IProdutoRepository produtoRepository)
8       {
9           _produtoRepository = produtoRepository;
10      }
11
        0 referências
12      public ResultMessage<Produto> Inserir(Produto produto)
13      {
14          if (produto.Validar())
15              return new ResultMessage<Produto>(false, "Campos Invalidos", null);
16
17          _produtoRepository.Insert(produto);
18
19          return new ResultMessage<Produto>(true, "Campos Invalidos", produto);
20      }
21
22  }
```

Fonte: De autoria própria.

Ao utilizar a inversão de dependência as classes passam a ser desacopladas da implementação das suas dependências, podendo elas serem modificadas sem as alterações refletirem diretamente nas classes que a utilizam. Ao usar interfaces tem-se a possibilidade de injetar diferentes implementações para a classe conforme

a necessidade, apenas indicando que a classe injetada deve implementar os métodos contidos na interface.

A utilização do DIP traz algumas melhorias, na solução apresentada o controller passa a estar completamente independente da implementação do repositório, e ao instanciar a classe ProdutoController pode-se injetar uma implementação de repositório para qualquer tipo de banco de dados, de acordo com a necessidade. Dessa forma melhora tanto a modificabilidade quanto a modularidade do sistema, tornando mais fácil modificar em certo nível os comportamentos sem alterar classes desnecessárias, e por colocar a interface na frente da implementação torna mais clara as dependências das classes que irão utilizar esta implementação.

O DIP junto com o SRP contribuem no melhor nível para o quesito de testabilidade, com a utilização do SRP torna-se mais claro o que cada trecho de código faz, mas ainda tem áreas na qual é custoso forçar determinado comportamento. Ao utilizar interfaces nas dependências das classes pode-se criar classes que simulam o comportamento esperado, injetá-las na classe principal e obter o cenário que deseja testar, com isso facilitando que esses testes explorem todos os cenários.

4.6. Análise Geral

Como observado, justificando a utilização em conjunto, cada um dos princípios não só definem aspectos a serem seguidos como resolvem problemas que os demais quando utilizados individualmente podem gerar.

Os conceitos são utilizados em conjunto mas podem e devem ser entendidos individualmente, para facilitar a compreensão e para a análise ser realizada e os pontos corrigidos na classe a ser implementada. A depender da forma utilizada para suprir algum princípio, outros arquivos podem ser criados, e a análise também deve ocorrer em cada um deles.

Implementar o princípio da responsabilidade única aumenta a coesão mas causa alto acoplamento, para remediar o acoplamento deve-se seguir o princípio da inversão de dependências que torna necessário a adoção de interfaces. As interfaces devem ser encaradas como uma via de mão dupla, classes que

implementam a interface se comprometem a cumpri-la, e classes que utilizam expõem a necessidade da execução das ações.

Ao escrever interfaces para fazer a inversão de dependências é necessário o cuidado para não quebrar os contratos e deve-se seguir o princípio de segregação de interfaces para ajudar na análise.

Na solução utilizada a ser adotada para cumprir o OCP, pode ser necessário criação de interfaces, outras classes, classes abstratas e as mesmas devem seguir os princípios também.

Resumo dos resultados obtidos para cada princípio:

- SRP: Auxilia na coesão, adverte sobre a separação de responsabilidades para classes e métodos.
- OCP: Auxilia na legibilidade do código e na manutenção ao alertar para o número de modificações que ocorrem e a possibilidade de extensão.
- LSP: Ajuda no mal funcionamento ao trazer a atenção para as heranças, e se realmente deveriam acontecer.
- ISP: Alerta a necessidade de criação de interfaces específicas para cada cenário, também evitando o mal funcionamento e trazendo o uso correto da funcionalidade da orientação a objetos.
- DIP: Resolve o alto acoplamento causado pelo SRP, possibilitando substituições de classes necessárias para o funcionamento e testes com cenários controlados.

5 - Conclusão

Como foi observado ao longo dos resultados, cada um dos princípios traz benefícios em diferentes aspectos, apesar da implementação ter sido realizada para analisar de forma isolada, a metodologia proposta para o bom emprego do SOLID contempla sua aplicação em conjunto.

Os princípios SRP, OCP, e LSP são focados em apresentar melhorias nas implementações, como melhor separação de responsabilidade das classes e métodos, permitindo a reutilização e acabando com a duplicidade, possibilita a extensão do comportamento sem alterações em diversos arquivos, e a substituição correta de classes pais e filhas sem corromper o comportamento esperado.

O princípio DIP orienta a depender de abstrações e soluciona o acoplamento que o SRP causa, a utilização de interfaces faz com que dependa apenas da existência das mesmas e torna explícito a necessidade de certos parâmetros e retornos nas classes que a implementam e utilizam.

Ao começar a criar interfaces, com o ISP pode-se trazer a preocupação com as responsabilidades para o cenário de interfaces, passando a escrever interfaces específicas para cada conjunto de comportamento esperado, para que possam ser utilizadas sem o conhecimento do código de implementação e com certeza de que todos os métodos terão utilidade.

A aplicação do SOLID não é tão custosa após o entendimento do conceito por trás de cada princípio e traz benefícios principalmente no longo prazo, não sendo necessário seguir absolutamente em todos os cenários mas ficando evidente a possibilidade de crescimento que os princípios trazem.

Analisando os pontos da ISO/IEC 25010, a aplicação do SOLID possibilita um aumento da reusabilidade em toda a aplicação, diminuindo a duplicidade. Por as classes e métodos se tornarem específicos, passa a ser mais simples analisar o software como um todo, encontrar defeitos ou mesmo compreender o que um trecho faz. O aumento da modularidade e a facilitação de modificações por permitir alterar métodos sem interferência direta em diversos códigos e a inclusão de novos comportamentos sem alterações desnecessárias ou repetidas em diversos arquivos. Pelas classes e métodos se tornarem mais específicos para cada comportamento esperado e a simulação de diferentes cenários ser mais simples pela utilização de interfaces, aumenta-se a testabilidade e torna mais fácil cobrir todo o código, facilitando assim a localização de erros através dos testes.

Assim foi possível averiguar como o SOLID contribui para a qualidade de software. Abrir mão dos princípios significa abrir mão da qualidade, aumentando a probabilidade de se obter o que se conhece como aplicação legado, com alterações sendo cada vez mais difíceis de serem realizadas. É essencial prestar o devido cuidado desde o início, pois adequar diversas classes posteriormente é muito custoso, já a análise após o aprendizado dos conceitos pode acontecer de forma quase natural.

Referências

Deitel, Harvey M. et al. *C# Como programar*. Pearson Education, 2003.

Eckel, Bruce. *Thinking in Java.* 4. ed. Prentice Hall. 2006.

ISO/IEC 25010. Março de 2011. Disponível em: <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>. Acesso em: 30 de maio de 2022.

MARTIN, Robert Cecil. *Design Principles and Design Patterns*, 2000. Disponível em: <https://web.archive.org/web/20150924054349/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf>. Acesso em: 15 de abril de 2022.

MARTIN, Robert Cecil *The Principles of OOD*. butUncleBob, 2005. Disponível em: <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>. Acesso em: 22 de março de 2022.

MARTIN, Robert Cecil. *The Single Responsibility Principle*. cleanCoder, 2014. Disponível em: <https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html> Acesso em: 22 de março de 2022.

PRESSMAN, Roger S; MAXIM, Bruce R. *Engenharia de Software: Uma abordagem profissional*. 8. ed. AMGH Editora, 2016.

THOMAS, David; HUNT, Andrew. *The Pragmatic Programmer*. P1.0. Pearson Education Inc, 2019.

APÊNDICE A - SRP - PRINCÍPIO DA RESPONSABILIDADE ÚNICA

1 - Violação:

1.1. Classe Pessoa.

```
namespace PrincipioResponsabilidadeUnica.Violacao
{
    3 referências
    public class Pessoa
    {
        2 referências
        public int Id { get; set; }
        2 referências
        public string Nome { get; set; }
        1 referência
        public string Email { get; set; }
        2 referências
        public DateTime DataNascimento { get; set; }
        2 referências
        public string CPF { get; set; }
        0 referências
        public Pessoa(int id, string nome, string email, DateTime dataNascimento, string cPF)
        {
            Id = id;
            Nome = nome;
            Email = email;
            DataNascimento = dataNascimento;
            CPF = cPF;
        }
    }
}

```

1.2. Classe PessoaService.

```
using System.Data.SqlClient;

namespace PrincipioResponsabilidadeUnica.Violacao
{
    1 referência
    public class PessoaService
    {
        private readonly SqlConnection _sqlconn;
        private const string _connectionString = "DATABASECONNECTIONSTRING";

        0 referências
        public PessoaService()
        {
            _sqlconn = new SqlConnection(_connectionString);
        }

        0 referências
        public Pessoa? InserirPessoa(Pessoa pessoa)
        {
            if (String.IsNullOrEmpty(pessoa.Nome))
                return null;
            if (ValidarCPF(pessoa.CPF))
                return null;
            if (pessoa.DataNascimento.AddYears(-18) > DateTime.Today.AddYears(-18))
                return null;

            _sqlconn.Open();

            var sqlCommand = new SqlCommand(
                $"INSERT INTO pessoa " +
                "(Nome, Email, DataNascimento, CPF)" +
                "VALUES" +
                "({pessoa.Nome}, {pessoa.Email}, {pessoa.DataNascimento}, {pessoa.CPF})"
            );

            var sqlData = sqlCommand.ExecuteReader();

            _sqlconn.Close();

            if (sqlData.RecordsAffected > 0)
            {
                while (sqlData.Read())
                {
                    pessoa.Id = (int)sqlData["Id"];
                }

                return pessoa;
            }

            return null;
        }
```

```
49
                1 referência
50              public bool ValidarCPF(string valor)
51              {
52                  int[] multiplicador1 = new int[9] { 10, 9, 8, 7, 6, 5, 4, 3, 2 };
53                  int[] multiplicador2 = new int[10] { 11, 10, 9, 8, 7, 6, 5, 4, 3, 2 };
54                  string tempCpf;
55                  string digito;
56                  int soma;
57                  int resto;
58                  if (valor.Length != 11)
59                      return false;
60                  tempCpf = valor.Substring(0, 9);
61                  soma = 0;
62
63                  for (int i = 0; i < 9; i++)
64                      soma += int.Parse(tempCpf[i].ToString()) * multiplicador1[i];
65                  resto = soma % 11;
66                  if (resto < 2)
67                      resto = 0;
68                  else
69                      resto = 11 - resto;
70                  digito = resto.ToString();
71                  tempCpf = tempCpf + digito;
72                  soma = 0;
73                  for (int i = 0; i < 10; i++)
74                      soma += int.Parse(tempCpf[i].ToString()) * multiplicador2[i];
75                  resto = soma % 11;
76                  if (resto < 2)
77                      resto = 0;
78                  else
79                      resto = 11 - resto;
80                  digito = digito + resto.ToString();
81                  return valor.EndsWith(digito);
82              }
83          }
84      }
85
```

2 - Solução:

2.1. Classe Pessoa.

```
1   namespace PrincipioResponsabilidadeUnica.Solucao
2   {
                4 referências
3       public class Pessoa
4       {
                    2 referências
5           public int Id { get; set; }
                    2 referências
6           public string Nome { get; set; }
                    1 referência
7           public string Email { get; set; }
                    2 referências
8           public DateTime DataNascimento { get; set; }
                    2 referências
9           public Cpf CPF { get; set; }
                    0 referências
10          public Pessoa(int id, string nome, string email, DateTime dataNascimento, Cpf cpf)
11          {
12              Id = id;
13              Nome = nome;
14              Email = email;
15              DataNascimento = dataNascimento;
16              CPF = cpf;
17          }
18
                    1 referência
19          public bool Validar()
20          {
21              if (String.IsNullOrEmpty(Nome))
22                  return false;
23              if (CPF.Validar())
24                  return false;
25              if (DataNascimento.AddYears(-18) > DateTime.Today.AddYears(-18))
26                  return false;
27
28              return true;
29          }
30
31      }
32  }
33
```

2.2. Classe PessoaService.

```
namespace PrincipioResponsabilidadeUnica.Solucao
{
    1 referência
    public class PessoaService
    {
        private readonly PessoaRepository _repository;

        0 referências
        public PessoaService()
        {
            _repository = new PessoaRepository();
        }

        0 referências
        public Pessoa? InserirPessoa(Pessoa pessoa)
        {
            if (pessoa.Validar())
                return null;

            _repository.InserirPessoa(pessoa);

            return pessoa;
        }

    }
}
```

2.3. Classe PessoaRepository.

```
using System.Data.SqlClient;

namespace PrincipioResponsabilidadeUnica.Solucao
{
    3 referências
    public class PessoaRepository
    {
        private readonly SqlConnection _sqlconn;
        private const string _connectionString = "DATABASECONNECTIONSTRING";

        1 referência
        public PessoaRepository()
        {
            _sqlconn = new SqlConnection(_connectionString);
        }

        1 referência
        public void InserirPessoa(Pessoa pessoa)
        {
            _sqlconn.Open();

            var sqlCommand = new SqlCommand(
                $"INSERT INTO pessoa " +
                "(Nome, Email, DataNascimento, CPF)" +
                "VALUES" +
                "({pessoa.Nome}, {pessoa.Email}, {pessoa.DataNascimento}, {pessoa.CPF})"
            );

            var sqlData = sqlCommand.ExecuteReader();

            while (sqlData.Read())
            {
                pessoa.Id = (int)sqlData["Id"];
            }

            _sqlconn.Close();
        }

    }
}
```

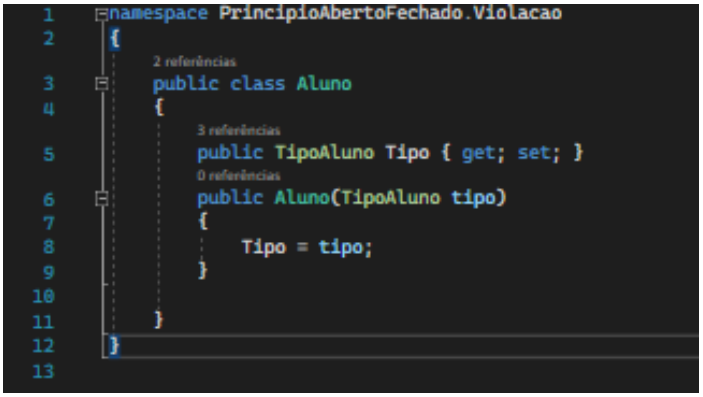
2.4. Classe CPF.

```
namespace PrincipioResponsabilidadeUnica.Solucao
{
    3 referências
    public class Cpf
    {
        4 referências
        public string Valor { get; set; }
        0 referências
        public Cpf(string valor)
        {
            Valor = valor;
        }

        1 referência
        public bool Validar()
        {
            int[] multiplicador1 = new int[9] { 10, 9, 8, 7, 6, 5, 4, 3, 2 };
            int[] multiplicador2 = new int[10] { 11, 10, 9, 8, 7, 6, 5, 4, 3, 2 };
            string tempCpf;
            string digito;
            int soma;
            int resto;
            if (Valor.Length != 11)
                return false;
            tempCpf = Valor.Substring(0, 9);
            soma = 0;

            for (int i = 0; i < 9; i++)
                soma += int.Parse(tempCpf[i].ToString()) * multiplicador1[i];
            resto = soma % 11;
            if (resto < 2)
                resto = 0;
            else
                resto = 11 - resto;
            digito = resto.ToString();
            tempCpf = tempCpf + digito;
            soma = 0;
            for (int i = 0; i < 10; i++)
                soma += int.Parse(tempCpf[i].ToString()) * multiplicador2[i];
            resto = soma % 11;
            if (resto < 2)
                resto = 0;
            else
                resto = 11 - resto;
            digito = digito + resto.ToString();
            return Valor.EndsWith(digito);
        }
    }
}
```

APÊNDICE B - OCP - PRINCÍPIO ABERTO/FECHADO

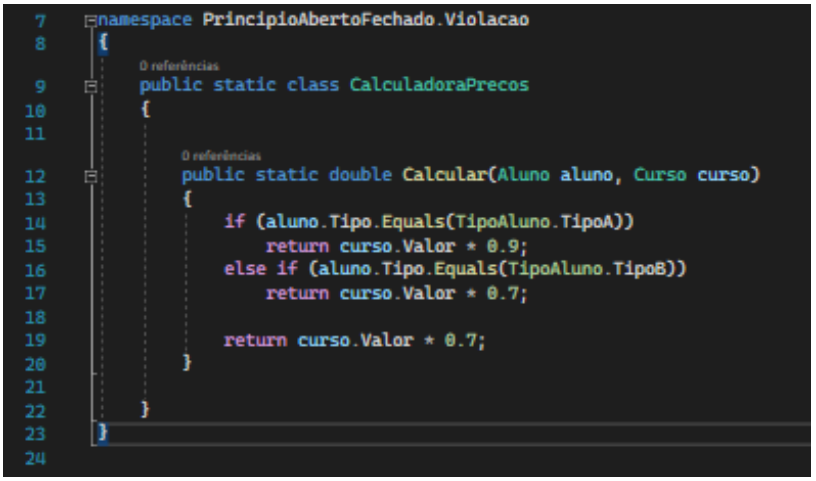
1 - Violação:

1.1. Classe Aluno.



```
namespace PrincipioAbertoFechado.Violacao
{
    2 referências
    public class Aluno
    {
        3 referências
        public TipoAluno Tipo { get; set; }
        0 referências
        public Aluno(TipoAluno tipo)
        {
            Tipo = tipo;
        }

    }
}
```

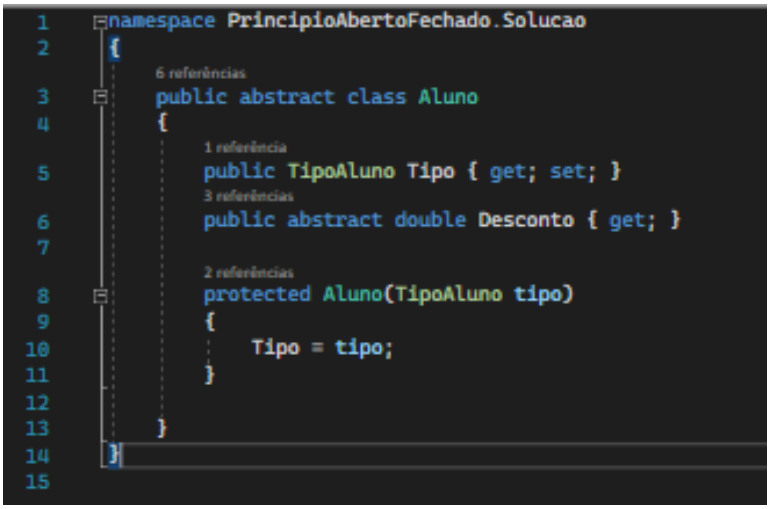
1.2. Classe CalculadoraPrecos.



```
namespace PrincipioAbertoFechado.Violacao
{
    0 referências
    public static class CalculadoraPrecos
    {

        0 referências
        public static double Calcular(Aluno aluno, Curso curso)
        {
            if (aluno.Tipo.Equals(TipoAluno.TipoA))
                return curso.Valor * 0.9;
            else if (aluno.Tipo.Equals(TipoAluno.TipoB))
                return curso.Valor * 0.7;

            return curso.Valor * 0.7;
        }

    }
}
```

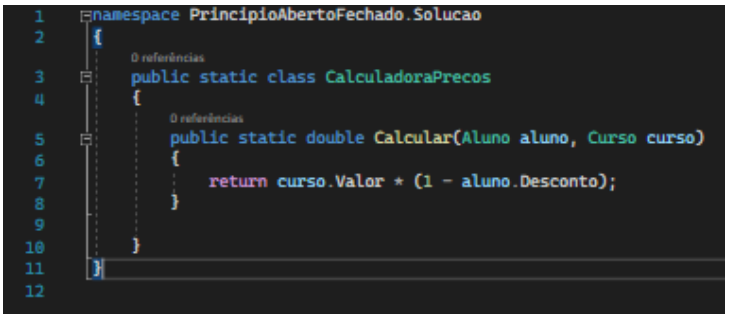
2 - Solução:

2.1. Classe Aluno.



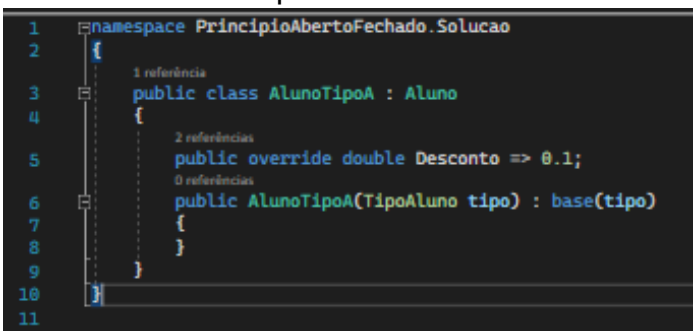
```
namespace PrincipioAbertoFechado.Solucao
{
    6 referências
    public abstract class Aluno
    {
        1 referência
        public TipoAluno Tipo { get; set; }
        3 referências
        public abstract double Desconto { get; }

        2 referências
        protected Aluno(TipoAluno tipo)
        {
            Tipo = tipo;
        }

    }
}
```

2.2. Classe CalculadoraPrecos.



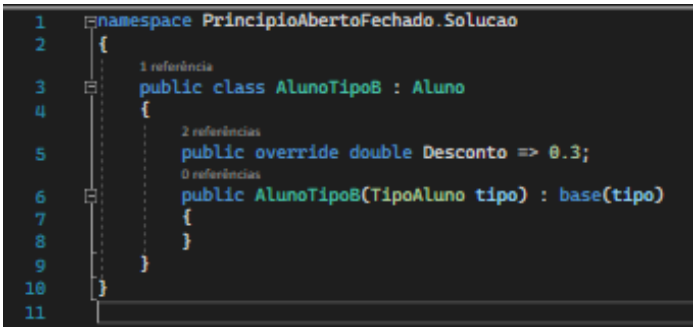
```
namespace PrincipioAbertoFechado.Solucao
{
    0 referências
    public static class CalculadoraPrecos
    {
        0 referências
        public static double Calcular(Aluno aluno, Curso curso)
        {
            return curso.Valor * (1 - aluno.Desconto);
        }

    }
}
```

2.3. Classe AlunoTipoA.



```
namespace PrincipioAbertoFechado.Solucao
{
    1 referência
    public class AlunoTipoA : Aluno
    {
        2 referências
        public override double Desconto => 0.1;
        0 referências
        public AlunoTipoA(TipoAluno tipo) : base(tipo)
        {
        }
    }
}
```

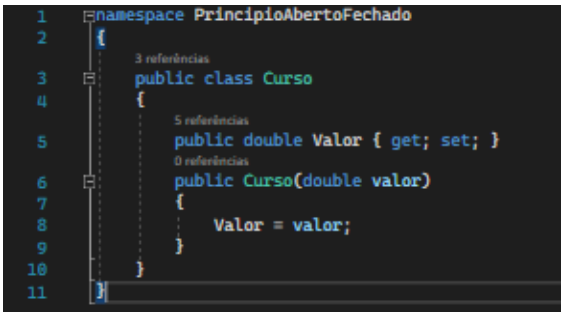
2.4. Classe AlunoTipoB.



```
namespace PrincipioAbertoFechado.Solucao
{
    1 referência
    public class AlunoTipoB : Aluno
    {
        2 referências
        public override double Desconto => 0.3;
        0 referências
        public AlunoTipoB(TipoAluno tipo) : base(tipo)
        {
        }
    }
}
```

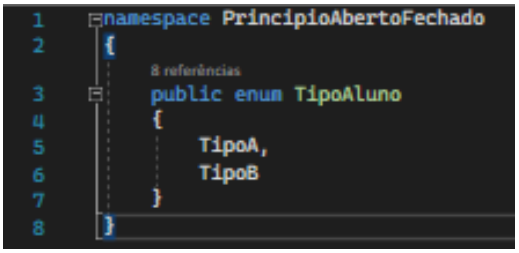
3 - Arquivos em comum:

3.1. Classe Curso.



```
namespace PrincipioAbertoFechado
{
    3 referências
    public class Curso
    {
        5 referências
        public double Valor { get; set; }
        0 referências
        public Curso(double valor)
        {
            Valor = valor;
        }
    }
}
```

3.2. Enum TipoAluno.



```
namespace PrincipioAbertoFechado
{
    8 referências
    public enum TipoAluno
    {
        TipoA,
        TipoB
    }
}
```

APÊNDICE C - LSP - PRINCÍPIO DE SUBSTITUIÇÃO DE LISKOV

1 - Violação:

1.1. Classe Gerente.

```
namespace PrincipioSubstituicaoLiskov.Violacao
{
    1 referência
    public class Gerente : Funcionario
    {
        0 referências
        public Gerente(double salario) : base(salario)
        {
        }

        0 referências
        public double Bonus()
        {
            return Salario * 0.2;
        }

    }
}
```

2 - Solução:

2.1. Classe Gerente.

```
namespace PrincipioSubstituicaoLiskov.Solucao
{
    1 referência
    public class Gerente
    {
        2 referências
        public double Salario { get; set; }
        0 referências
        public Gerente(double salario)
        {
            Salario = salario;
        }

        0 referências
        public double Bonus()
        {
            return Salario * 0.2;
        }

    }
}
```

3 - Arquivos em comum:

3.1. Classe Funcionario.

```
namespace PrincipioSubstituicaoLiskov
{
    3 referências
    public class Funcionario
    {
        2 referências
        public double Salario { get; set; }
        3 referências
        public IEnumerable<DateTime> Pontos { get; set; }
        1 referência
        public Funcionario(double salario)
        {
            Salario = salario;
            Pontos = new List<DateTime>();
        }
        0 referências
        public void RegistrarPonto(DateTime ponto)
        {
            Pontos = Pontos.Append(ponto);
        }

    }
}
```

APÊNDICE D - ISP - PRINCÍPIO DE SEGREGAÇÃO DE INTERFACES

1 - Violação:

1.1. Interface IProdutoService.

```
namespace PrincipioSegregacaoInterface.Violacao
{
    2 referências
    public interface IProdutoService
    {
        2 referências
        void ValidarRequisitos();
        2 referências
        void RemoverEstoque();
        2 referências
        void RealizarEnvio();
    }
}
```

1.2. Classe ProdutoService.

```
namespace PrincipioSegregacaoInterface.Violacao
{
    0 referências
    public class ProdutoService : IProdutoService
    {
        1 referência
        public void ValidarRequisitos()
        {
            //VALIDA REQUISITOS PARA PRODUTO COMUM
        }

        1 referência
        public void RemoverEstoque()
        {
            //REMOVE DO ESTOQUE O PRODUTO
        }

        1 referência
        public void RealizarEnvio()
        {
            //REALIZA O ENVIO PARA O ENDEREÇO DO COMPRADOR
        }

    }
}
```

1.3. Classe ProdutoDigitalService.

```
namespace PrincipioSegregacaoInterface.Violacao
{
    0 referências
    public class ProdutoDigitalService : IProdutoService
    {
        1 referência
        public void ValidarRequisitos()
        {
            //VALIDA REQUISITOS PARA PRODUTO DIGITAL
        }

        1 referência
        public void RemoverEstoque()
        {
            throw new ArgumentException("PRODUTO DIGITAL NÃO TEM ESTOQUE");
        }

        1 referência
        public void RealizarEnvio()
        {
            //REALIZA O ENVIO PARA O EMAIL DO COMPRADOR
        }

    }
}
```

2 - Solução:

2.1. Interface IProdutoService.

```
namespace PrincipioSegregacaoInterface.Solucao
{
    1 referência
    public interface IProdutoService
    {
        1 referência
        void RemoverEstoque();
        1 referência
        void ValidarRequisitos();
        1 referência
        void RealizarEnvio();
    }
}
```

2.2. Interface IProdutoDigitalService.

```
namespace PrincipioSegregacaoInterface.Solucao
{
    1 referência
    public interface IProdutoDigitalService
    {
        1 referência
        void ValidarRequisitos();
        1 referência
        void RealizarEnvio();
    }
}
```

2.3 Classe ProdutoService.

```
namespace PrincipioSegregacaoInterface.Solucao
{
    1 referência
    public interface IProdutoService
    {
        1 referência
        void RemoverEstoque();
        1 referência
        void ValidarRequisitos();
        1 referência
        void RealizarEnvio();
    }
}
```

2.4 Classe ProdutoDigitalService.

```
namespace PrincipioSegregacaoInterface.Solucao
{
    0 referências
    public class ProdutoDigitalService : IProdutoDigitalService
    {
        1 referência
        public void ValidarRequisitos()
        {
            //VALIDA REQUISITOS PARA PRODUTO DIGITAL
        }

        1 referência
        public void RealizarEnvio()
        {
            //REALIZA O ENVIO PARA O EMAIL DO COMPRADOR
        }

    }
}
```

APÊNDICE E - DIP - PRINCÍPIO DA INVERSÃO DE DEPENDÊNCIAS

1 - Violação:

1.1. Classe ProdutoController.

```
namespace PrincipioInversaoDependencia.Violacao
{

    public class ProdutoController
    {
        private readonly ProdutoService _produtoService;


        public ProdutoController()
        {
            _produtoService = new ProdutoService();
        }

        0 referências
        public ResultMessage<Produto> Post(Produto produto)
        {
            if (produto.Validar())
                return new ResultMessage<Produto>(false, "Campos Invalidos", null);

            _produtoService.InserirProduto(produto);

            return new ResultMessage<Produto>(true, "Campos Invalidos", produto);
        }

    }
}
```

1.2. Classe ProdutoService.

```
namespace PrincipioInversaoDependencia.Violacao
{
    3 referências
    public class ProdutoService
    {
        1 referência
        public ProdutoService()
        {
        }

        1 referência
        public void InserirProduto(Produto produto)
        {
            //CÓDIGO RESPONSÁVEL PELA INSERÇÃO
        }
    }
}
```

2 - Solução:

2.1. Classe ProdutoController.

```
namespace PrincipioInversaoDependencia.Solucao
{
    1 referência
    public class ProdutoController
    {
        private readonly IProdutoService _produtoService;

        0 referências
        public ProdutoController(IProdutoService produtoService)
        {
            _produtoService = produtoService;
        }

        0 referências
        public ResultMessage<Produto> Post(Produto produto)
        {
            if (produto.Validar())
                return new ResultMessage<Produto>(false, "Campos Invalidos", null);

            _produtoService.InserirProduto(produto);

            return new ResultMessage<Produto>(true, "Campos Invalidos", produto);
        }

    }
}
```

2.2. Interface IProdutoService.

```
namespace PrincipioInversaoDependencia.Solucao
{
    2 referências
    public interface IProdutoService
    {
        1 referência
        void InserirProduto(Produto produto);
    }
}
```

3 - Arquivos em comum:

3.1. Classe Produto.

```
namespace PrincipioInversaoDependencia
{
    11 referências
    public class Produto
    {
        1 referência
        public int Id { get; set; }
        2 referências
        public string Nome { get; set; }
        2 referências
        public double Preco { get; set; }

        0 referências
        public Produto(int id, string nome, double preco)
        {
            Id = id;
            Nome = nome;
            Preco = preco;
        }

        2 referências
        public bool Validar()
        {
            if (String.IsNullOrEmpty(Nome))
                return false;
            if (String.IsNullOrEmpty(Preco))
                return false;

            return true;
        }
    }
}
```

3.2. Classe ResultMessage.

```
namespace PrincipioInversaoDependencia
{
    7 referências
    public class ResultMessage<T>
    {
        1 referência
        public bool Result { get; set; }
        1 referência
        public string Message { get; set; }
        1 referência
        public T? Data { get; set; }

        4 referências
        public ResultMessage(bool result, string message, T? data)
        {
            Result = result;
            Message = message;
            Data = data;
        }

    }
}
```