INTERFACE TECNOLÓGICA

PRINCÍPIO DA RESPONSABILIDADE ÚNICA APLICADA AO DESENVOLVIMENTO DE APLICAÇÕES

SINGLE RESPONSABILITY PRINCIPLE APPLIED TO THE DEVELOPMENT OF APPLICATIONS

Ester Andrade Ribeiro – ester.andrade2017@outlook.com

Fernando Tiosso - fernando.tiosso@fatectq.edu.br

Faculdade de Tecnologia de Taquaritinga (FATEC) – SP - Brasil

RESUMO

O uso de *designs patterns* no desenvolvimento de *softwares* está cada vez mais em evidência no mercado profissional. Sua utilização possibilita, além da fácil manutenção da aplicação, uma padronização e reutilização de componentes já desenvolvidos, proporcionando agilidade na implementação de novas funcionalidades. Visando a investigação do uso de *designs patterns*, este artigo apresenta, por meio de uma pesquisa bibliográfica e um estudo de caso prático, a importância da utilização de um dos princípios, conhecido como o princípio da responsabilidade única, referente a um padrão de projeto difundido no mundo da codificação: o *SOLID.* O princípio da responsabilidade única possui grande importância dentro da estrutura de uma aplicação, visto que a implementação de um *software* sem o seu suporte pode resultar em um produto de baixa qualidade, difícil manutenção e com problemas de performance. Portanto, com a adequada utilização deste primeiro princípio, não somente se alcança uma codificação "mais limpa", mas também se alcança qualidade em termos de processos executados pelo *software*, unidos com uma notável divisão e melhor estruturação funcional do projeto. Pode-se concluir ainda que a utilização do primeiro princípio não deve ser entendida como "modismo", mas como uma padronização necessária que deve ser aplicada a um projeto de *software*, visando o aumento da qualidade e um melhor entendimento dos códigos produzidos durante a fase da sua implementação.

Palavras-chave: Princípio da responsabilidade única. Padrões de projeto. *SOLID*. Qualidade de código.

ABSTRACT

The use of design patterns in the development of software is increasingly in evidence in the professional market. Besides making the maintenance of the application easier, it also allows the standardization and the re-use of already developed components, providing agility in the implementation of new functionalities. Aiming to understand the use of design patterns, this article presents, by means of a bibliographical research and a study of practical case, the importance of the use of one of the principles, known as the Single Responsibility Principle, concerning a standard of a widespread design pattern in the world of codification: the SOLID. The Single Responsibility Principle has a great importance in the structure of an application, since the implementation of a software without its support can result in a product of low quality, difficult maintenance and with performance problems. Therefore, with the appropriate

use of this first principle, not only a “cleaner” codification is reached, but also a quality in terms of the processes executed by the software, along with a notable division and a better functional structuring of it. Hence, the use of the first principle should not be understood as a “fad”, but as a necessary standardization that must be applied to a software project, aiming the increase of the quality and a better understanding of the produced code during the stage of its implementation.

Keywords: Single Responsibility Principle. Design Patterns. SOLID. Code Quality.

1 INTRODUÇÃO

A busca pela produção de um *software* de qualidade tem feito com que diversos desenvolvedores invistam e se preocupem não só com os fluxos a serem realizados por uma aplicação, mas também com o código que deve ser implementado para construí-los. A escolha da estrutura e dos padrões que serão seguidos durante o desenvolvimento do *software* são fatores que podem determinar a eficácia dos seus procedimentos, bem como a fácil manutenção ao longo do seu ciclo de vida.

O alcance desses benefícios está intimamente relacionado com a utilização de padrões, mais conhecidos como *design patterns.* De acordo com Leite (2005), estes padrões foram introduzidos por Christopher Alexander nos anos 70 e têm como objetivo a resolução de obstáculos comumente presentes nas atividades de codificação**.**

Neste contexto, Pressman (2011), afirma que cada *pattern* descreve um problema e propõe uma solução na qual o desenvolvedor pode aplicá-la diversas vezes em um mesmo projeto, resultando no reuso de código e descartando a ideia de se criar do zero, o que já foi desenvolvido.

Porém, Sommerville (2007), cita que este reuso não é simplesmente usar o que já está desenvolvido, para que isso aconteça de maneira correta é necessário haver um bom planejamento arquitetural tendo em vista manter a organização dentro do projeto e a aplicação correta dos padrões.

Levando em consideração a qualidade do código da aplicação, é de extrema importância dar atenção a estas melhores práticas de codificação, pois a elaboração de um *software* sem nenhum tipo de paradigma ou estudos de melhores caminhos pode resultar em um produto com baixíssima qualidade esperada e com sérios problemas de performance.

Para tanto, este artigo apresentará, por meio de uma revisão bibliográfica e um estudo de caso prático, o primeiro princípio, mais conhecido como princípio de responsabilidade única, do *design pattern* bem difundido entre os desenvolvedores de *software*: o *SOLID*.

Assim, o corrente artigo foi dividido em 5 (cinco) seções, sendo estas: a seção 2 (dois) responsável por apresentar as metodologias de pesquisa que foram utilizadas em sua elaboração, a seção 3 (três) responsável pela fundamentação teórica a respeito dos padrões de projeto, conceitos da programação orientada a objetos e o primeiro princípio do conjunto de princípios conhecidos como *SOLID*, a seção 4 (quatro) encarregada pela demonstração da infração e correção do princípio da responsabilidade única por meio de um estudo de caso e, por fim, a seção 5 (cinco) responsável por apresentar as conclusões a respeito deste trabalho.

2 METODOLOGIA DE PESQUISA

Para a construção deste trabalho foi necessário utilizar-se do método de pesquisa bibliográfica. A esse respeito, Lakatos (2009) declara:

> Trata-se de um levantamento de toda a bibliografia já publicada, em forma de livros, revistas, publicações avulsas e imprensa escrita. Sua finalidade é colocar o pesquisador em contato direto com tudo aquilo que foi escrito sobre determinado assunto (p.43).

Além disso, também foi utilizado o método de estudo de caso prático que tem como objetivo explicar algumas questões norteadoras sobre uma determinada pesquisa, sendo o "como" e o "por quê" apresentados. Segundo Yin (2005, p.32), um estudo de caso pode ser compreendido como sendo "uma investigação empírica que investiga um fenômeno contemporâneo dentro de seu contexto da vida real, especialmente quando os limites entre o fenômeno e o contexto não estão claramente definidos ".

Portanto, o desenvolvimento do trabalho contou com a utilização de ambas as metodologias, sendo utilizadas em sequência para a sua estruturação e elaboração. Sendo assim, a seção subsequente apresenta, através do método de revisão bibliográfica, toda a fundamentação teórica que sustenta este trabalho.

3 FUNDAMENTAÇÃO TEÓRICA

3.1 Padrões de projeto

De acordo com Leite (2005), os padrões de projeto são algumas ideias que foram apresentadas inicialmente por Christopher Alexander nos anos de 1977 afim de criar soluções para problemas comuns entre projetos. Sommerville (2007, p.279), apresenta a seguinte

definição para o termo padrão, “o padrão é uma descrição do problema e a essência da sua solução, dessa forma a solução pode ser reusada em aplicações diferentes”. Mediante esta definição apresentada pelo autor, pode-se concluir que este conjunto de paradigmas tem como principal objetivo propor soluções com teorias comprovadas, por meio de melhores práticas, para problemas correntes e específicos entre desenvolvedores.

Frequentemente, os padrões de um projeto são definidos no início do mesmo e documentados para que todos os membros da equipe tenham conhecimento sobre eles. Porém, Pressman (2011), cita que uma das etapas mais difíceis para todos os indivíduos envolvidos em um sistema é a etapa responsável pela busca e definição dos padrões que realmente atendam à sua necessidade.

Boa parte dos *designs patterns* existentes hoje estão ligados aos conceitos da programação orientada a objetos, como, por exemplo, os princípios de *SOLID*. É de grande importância que os membros da equipe de desenvolvimento tenham os conhecimentos básicos de orientação a objetos para que seja possível sua aplicabilidade de maneira correta. Portanto, o item subsequente, traz os conceitos básicos da programação orientada a objetos que devem ser compreendidos pela equipe de desenvolvimento do projeto.

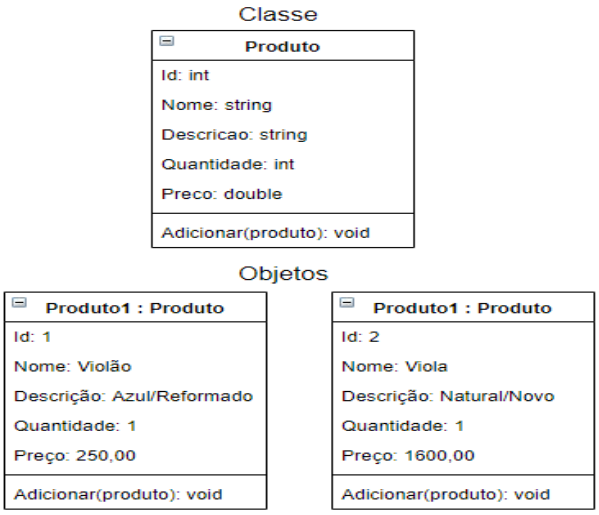
3.2 Conceitos básicos da programação orientada a objetos

3.2.1 Classes

Para se conseguir entender o conceito de uma classe é necessário levar em consideração o primeiro pilar da programação orientada a objetos, a abstração. De acordo com Furlan (1998), uma classe é definida como uma abstração de um grupo de objetos parecidos, geralmente caracterizadas como lugares, tipos de pessoas ou objetos que compõe o uso dos mesmos atributos e operações.

As classes possuem declarações de atributos e métodos e cada objeto criado a partir delas podem receber valores em suas características (atributos), bem como fazer uso dos métodos propostos. Conforme mostra a Figura 1, a classe “Produto” possui os atributos: “Id”, “Nome”, “Descrição”, “Quantidade” e “Preço”.

Figura 1: Diagrama de classes e objetos



Fonte: Elaborado pelos autores (2019)

Ao ser criado um objeto, uma nova instancia a partir de uma classe, os atributos são inicializados para que possam receber novos valores e os métodos ficam disponíveis para que possam ser invocados. Ainda observando-se a Figura 1, é possível notar o método "Adicionar", que pode ser invocado por qualquer objeto instanciado a partir da classe Produto.

O termo "objeto", citado acima, é outro item que deve ser compreendido em meio aos conceitos da programação orientada a objetos e, para tanto, será apresentado e explicado a seguir.

3.2.2 Objetos

Segundo Pressman (2011, p.163), um objeto pode ser definido como sendo "uma representação das informações compostas que devem ser compreendidas pelo *software*. Por informação composta, entende-se algo que tenha uma série de propriedades ou atributos diferentes". Já Vincenzi (2004, p.10), defini um objeto como sendo "uma instância de uma classe criada em tempo de execução. Cada objeto tem uma cópia dos dados definidos na classe".

Diante dessas afirmações, diferentes valores podem ser atribuídos para os atributos que foram definidos na classe da qual o objeto foi criado, conforme mostra a Figura 1. Desta

forma, constata-se que os atributos são fundamentais para a criação dos objetos e seus conceitos serão apresentados no item subsequente.

3.2.3 Atributos

Segundo Rumbaugh (1994), os atributos são "locais" que permitem o armazenamento de valores específicos para as características existentes nos objetos criados a partir de uma classe. Segundo a documentação oficial da Microsoft (2018), estes atributos conseguem fornecer, de uma melhor maneira, um produto e suas características, através de campos pré-definidos pelo próprio usuário, ou seja, eles auxiliam na demonstração dos dados do sistema, auxiliando em tomadas de decisões dentro do projeto, realização de consultas, cadastros, entre outros.

Um objeto pode ter N atributos e cada um deles pode receber os valores que foram informados pelo usuário, pré-definidos pelo desenvolvedor ou ainda calculados dentro da própria solução. Conforme ilustra a Figura 1, no objeto "Produto1" há o atributo "Nome" que tem seu valor definido pela palavra "Violão". Caso uma nova instância da classe "Produto" fosse criada, seria possível atribuir um novo valor para este atributo, bem como para todos os outros atributos existentes na classe "Produto".

No entanto, além dos atributos, um objeto pode conter métodos que afetam seu comportamento e/ou mudam seu estado, explicados e detalhados a seguir.

3.2.4 Métodos

Segundo Rumbaugh (1994), os métodos são operações e/ou ações que os objetos podem executar, caracterizadas como funções. Um cachorro, por exemplo, pode ter as seguintes funções: andar; correr; latir e cada uma delas possui suas particularidades e sequências lógicas para que possam ser realizadas com sucesso. A Microsoft (2015) explana que cada método possui um determinado conjunto de código com uma sequência lógica de instruções.

Assim, um objeto pode executar diversos métodos de acordo com sua necessidade. Conforme mostra a Figura 1, o objeto "Produto1" possui atributos e métodos. Neste caso, a única operação que pode ser executada pelo objeto é a operação proporcionada pelo método "Adicionar".

Ressalta-se que os atributos de um objeto também podem ser encapsulados por métodos para garantir que recebam valores de acordo com as regras de negócio pré-estabelecidas no sistema. Desta forma, métodos são criados para manipular os valores contidos nos atributos e, com objetivo de manipular os valores dos atributos de um objeto mais facilmente, surgiram as propriedades, explicadas a seguir.

3.2.5 Propriedades

Segundo Camargo (2010), as propriedades podem ser entendidas como sendo estruturas presentes em uma classe que definem o acesso às suas informações, encapsulando seus atributos. Propriedades podem ser constituídas por dois conhecidos métodos: *get* e *set.*

O método *get* é responsável por retornar os valores armazenados nos atributos e o método *Set* pela atribuição desses valores. Esses métodos promovem o encapsulamento dos atributos e a inclusão de regras de negócios para validar os dados, antes que os mesmos sejam alocados nos atributos.

Com a base fundamental necessária para se compreender os principais conceitos de orientação a objetos e permitir a explicação do primeiro princípio de *SOLID*, o próximo item apresenta o conteúdo referente ao princípio da responsabilidade única.

3.3 Princípio da responsabilidade única – SRP

De acordo com Pires (2013), *SOLID* é um acrônimo de cinco princípios da programação orientada a objetos, introduzido por Robert C. Martin no livro “Princípios de Design e Padrões de Design”. O foco deste artigo é apresentar e discutir o primeiro princípio, *Single Responsability Principle* (SPR), que em português significa “Princípio da Responsabilidade Única”.

O princípio da responsabilidade única defende a ideia de que uma classe deve possuir apenas uma responsabilidade, isto é, realizar apenas uma tarefa ou ação, e tal conceito resulta na redução do acoplamento entre as classes. Com o objetivo de elucidar este princípio de forma mais clara, pode-se entender que uma classe deve ter exclusivamente um único motivo que justifique a sua mudança. Portanto, se uma classe foi projetada para enviar e-mails, ela irá apenas enviar e-mails e nada mais. Por outro lado, se uma classe foi projetada para salvar um novo funcionário no banco de dados, ela irá apenas salvar os dados do funcionário.

A fim de exemplificar esse princípio e sua aplicação prática, elaborou-se um estudo de caso prático para demonstrar o correto uso do mesmo, visando a reutilização e a redução de códigos, facilitando a manutenção e extensão do projeto por meio do fraco acoplamento entre as classes. Todo o estudo de caso será descrito no item seguinte.

4 RESULTADOS E DISCUSSÃO DO ESTUDO DE CASO

O presente estudo de caso analisou, de forma prática, a diferença do desenvolvimento de um código que não utiliza o primeiro princípio de *SOLID* e um código que o utiliza, por meio do framework .NET e da linguagem de programação C#. Esta pesquisa teve como objetivo exemplificar a maneira correta de se aplicar o princípio da responsabilidade única nos projetos de desenvolvimento de software.

A Figura 2 apresenta, incialmente, um exemplo da infração do princípio da responsabilidade única. De acordo com este princípio, o método “Adicionar” da classe produto deveria apenas adicionar um novo produto, porém, ele faz mais do que isso. Observando-o, tem-se a nítida impressão de que ele é um método de validação e inserção ao mesmo tempo, ou seja, ele possui mais de uma responsabilidade.

Figura 2: Infração do princípio da responsabilidade única

```
namespace SOLID.Infracao
{
    public class Produto
    {
        public int Id { get; set; }
        public string Nome { get; set; }
        public string Descricao { get; set; }
        public double Preco { get; set; }
        public int Quantidade { get; set; }

        public void Adicionar(Produto produto)
        {
            if (produto.Id == 0)
                Console.WriteLine("Não foi possível adicionar o produto pois o Id está vazio");
            else if (String.IsNullOrEmpty(produto.Nome))
                Console.WriteLine("Não foi possível adicionar o produto pois o nome está vazio");
            else if (String.IsNullOrEmpty(produto.Descricao))
                Console.WriteLine("Não foi possível adicionar o produto pois a descrição está vazio");
            else if (produto.Preco == 0)
                Console.WriteLine("Não foi possível adicionar o produto pois o preço está vazio");
            else if(produto.Quantidade == 0)
                Console.WriteLine("Não foi possível adicionar o produto pois a quantidade está vazia");
            else
                //salvar no banco de dados
                Console.WriteLine("Produto cadastrado!");
            Console.ReadKey();
        }
    }
}
```

Fonte: Elaborado pelos autores (2019)

Assim sendo, permitiu-se a criação de uma nova instancia da classe “Produto” e a invocação do método “Adicionar”, conforme mostra a Figura 3.

Figura 3: Trecho código infringindo o princípio da responsabilidade única

```
// Infração
Infracao.Produto produto = new Infracao.Produto();

produto.Id = 1;
produto.Nome = "Violão Takamine";
produto.Descricao = "Violão Folk Natural";
produto.Quantidade = 10;
produto.Preco = 2500.00;

produto.Adicionar(produto);
```

Fonte: Elaborado pelos autores (2019)

Todavia, seguindo o princípio da responsabilidade única, foi possível refatorar este código diminuindo seu acoplamento. Assim, a classe Produto passou a conter somente os atributos necessários, encapsulados por suas respectivas propriedades, para a definição de um produto, conforme mostra a Figura 4.

Figura 4: Classe produto com seus atributos

```
namespace SOLID.Correcao
{
    public class Produto
    {
        public int Id { get; set; }
        public string Nome { get; set; }
        public string Descricao { get; set; }
        public double Preco { get; set; }
        public int Quantidade { get; set; }
    }
}
```

Fonte: Elaborado pelos autores (2019)

Consequentemente, uma nova classe foi criada com a responsabilidade de implementar o método para adicionar um novo produto, conforme mostra a Figura 5.

Figura 5: Classe AdicionarProduto com seu método Adicionar

```
namespace SOLID.Correcao
{
    public class AdicionarProduto
    {

        public void Adicionar(Produto produto)
        {
            ValidarProduto validarProduto = new ValidarProduto();

            var result = validarProduto.ValidarCampos(produto);
            if(result.Count == 0)
                Console.WriteLine("Produto salvo!");
            else
            {
                Console.WriteLine("Não foi possível adicionar o produto pois : \n");
                foreach (var item in result)
                {
                    Console.WriteLine(item.Mensagem);
                }
            }
            Console.ReadKey();
        }
    }
}
```

Fonte: Elaborado pelos autores (2019)

Analisando a Figura 5, nota-se que o método “Adicionar” não faz algum tipo de validação nos atributos do objeto produto, pelo contrário, seguindo a ideia do princípio da responsabilidade única, foi necessário criar uma nova classe com a função de realizar todas as validações necessárias para que o produto fosse armazenado de forma consistente, conforme mostra a Figura 6.

Figura 6: **Classe ValidarProduto com o seu método ValidarCampos**

```
namespace SOLID.Correcao
{
    public class ValidarProduto
    {
        public List<MensagemErro> ValidarCampos(Produto produto)
        {
            List<MensagemErro> mensagensErro = new List<MensagemErro>();

            if (produto.Id == 0)
                mensagensErro.Add(new MensagemErro("O Id não pode ser nulo"));
            if (String.IsNullOrEmpty(produto.Nome))
                mensagensErro.Add(new MensagemErro("O nome não pode ser nulo"));
            if (String.IsNullOrEmpty(produto.Descricao))
                mensagensErro.Add(new MensagemErro("A descrição não pode ser nula"));
            if (produto.Preco == 0)
                mensagensErro.Add(new MensagemErro("O preço não pode ser nulo"));
            if (produto.Quantidade == 0)
                mensagensErro.Add(new MensagemErro("A quantidade não pode ser nula"));

            return mensagensErro;
        }
    }
}
```

Fonte: Elaborado pelos autores (2019)

Observando-se a Figura 6, constata-se que a classe “ValidarProduto” possui o método “ValidarCampos”, responsável por validar os atributos do produto e retornar uma lista de mensagens de erro, ou seja, a cada infração que for encontrada no objeto, uma mensagem explicativa será adicionada na lista de mensagens de erros. Ressalta-se que para adicionar mensagens de erro à lista de mensagens, foi necessário criar uma outra classe para permitir a inclusão de comunicados de erro conforme ilustra a Figura 7.

Figura 7: Classe MensagemErro

```
namespace SOLID.Correcao
{
    public class MensagemErro
    {
        public string Mensagem{ get; set; }

        public MensagemErro(string Mensagem)
        {
            this.Mensagem = Mensagem;
        }
    }
}
```

Fonte: Elaborado pelos autores (2019)

Após o refatoramento dos códigos, finalmente foi possível inserir um novo produto seguindo os padrões e regras estabelecidos pelo princípio da responsabilidade única, como mostra a Figura 8. Nela, é possível observar que uma nova instancia da classe “Produto” foi criada, permitindo a definição das características do produto e, logo em seguida, também foi criada uma nova instancia da classe “AdicionarProduto”, permitindo a persistência dos dados do produto de forma consistente por meio do método “Adicionar”.

Figura 8: Trecho código adequado ao princípio da responsabilidade única

```
//Correção
Correcao.Produto produto = new Correcao.Produto();
produto.Id = 1;
produto.Nome = "Violão Takamine";
produto.Descricao = "Violão Folk Natural";
produto.Quantidade = 10;
produto.Preco = 2500.00;

AdicionarProduto adicionarProduto = new AdicionarProduto();
adicionarProduto.Adicionar(produto);
```

Fonte: Elaborado pelos autores (2019)

Com base no experimento realizado, foi possível notar que o estudo de caso prático demostra que a implementação do primeiro princípio de *SOLID*, o princípio da responsabilidade única, requer um bom planejamento de quais classes serão necessárias dentro de sistema, ressaltando a importância de utiliza-las de forma desacoplada para que suas funções estejam bem divididas proporcionando uma melhor estruturação funcional do projeto. Assim, os resultados apresentados neste artigo concentraram-se em demostrar um exemplo de como dividir as responsabilidades das classes mediante seus processos e regras de negócio.

5 CONCLUSÃO

Realizando uma análise do uso de *design patterns* em meio ao ecossistema de desenvolvimento de software, especialmente do primeiro princípio de *SOLID* conhecido como o princípio da responsabilidade única, nota-se que a preocupação em criar e organizar a codificação de um sistema tem aumentado cada vez mais entre os membros de uma equipe, afim de garantir a qualidade do código visando sua manutenibilidade e futura extensão. No entanto, para que se consiga entender e implementar este princípio de maneira correta é de fundamental importância conhecer detalhadamente alguns conceitos da programação orientada a objetos, tais como: classes, objetos, atributos, métodos e propriedades. Neste caso, o zelo com a implementação do código está diretamente ligado a preocupação do bom

funcionamento dos processos de negócio valorizando suas questões de performance, bom entendimento, manutenibilidade e extensão, utilizando as melhores práticas estabelecidas e difundidas pelo meio acadêmico e/ou profissionais reconhecidos no mercado de trabalho profissional, para que no final do projeto seja possível obter um produto que atenda as reais necessidades do cliente e também agregue valor para a equipe/empresa responsável pelo desenvolvimento, após à realização de um considerável investimento tecnológico.

REFERÊNCIAS

CAMARGO, Wellington Balbo de. **Propriedades e Eventos – Classes**: Programação Orientada a Objetos – Parte 2. 2010. Disponível em: <https://www.devmedia.com.br/propriedades-e-eventos-classes-programacao-orientada-a-objetos-parte-2/18577>. Acesso em: 02 mar. 2019.

FURLAN, José Davi. **Modelagem de Objetos através da UML** – *the Unified Modeling Language* – São Paulo: Makron Books, 1998.

LAKATOS, Eva Maria. **Metodologia do trabalho científico**: procedimentos básicos, pesquisa bibliográfica, projeto e relatório, publicações e trabalhos científicos. 4. Ed – São Paulo: Atlas, 1992.

LEITE, Alessandro Ferreira. **Conheça os Padrões de Projeto**. 2005. Disponível em: <https://www.devmedia.com.br/conheca-os-padroes-de-projeto/957>. Acesso em: 22 fev. 2019.

MICROSOFT. **Métodos (Guia de Programação em C#)**. 2015. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/methods> Acessado em: 04 mar. 2019.

____. **Atributos (C#)**. 2018. Disponível em: < https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/attributes/> Acessado em: 04 mar. 2019.

PIRES, Eduardo. **SOLID – Teoria e Prática – Demo + Vídeo**. 2015. Disponível em: <https://www.eduardopires.net.br/2015/01/solid-teoria-e-pratica/> Acessado em: 12 mar. 2019.

PRESSMAN, Roger S. **Engenharia de software**: uma abordagem profissional. 7.Ed – Porto Alegre: AMGH, 2011.

RUMBAUGH, James. **Modelagem e projetos baseados em objetos.** Rio de Janeiro: Campus, 1994.

SOMMERVILLE, Ian. **Engenharia de software**, 8. Ed. São Paulo: Pearson Addison-Wesley, 2007.

VINCENZI, Auri Marcelo Rizzo. **Orientação a objeto**: definição, implementação e análise de recursos de teste e validação. 2004. Tese (Doutorado em Ciências de Computação e Matemática Computacional) – ICMC/USP/São Carlos, 2004.

YIN, Robert K. **Estudo de caso**: planejamento e métodos. 3. Ed – Porto Alegre: Bookman, 2005.