INTERFACE TECNOLÓGICA

CÓDIGO LIMPO: padrões e técnicas no desenvolvimento de software

CLEAN CODE: patterns and techniques in software development

Phelipe Alex Magalhães – phelipemagalhaes13@gmail.com
Fernando Tiosso – fernando.tiosso@fatectq.edu.br
Faculdade de Tecnologia de Taquaritinga (FATEC) – SP – Brasil

RESUMO

Com o grande número de códigos escritos diariamente pelos desenvolvedores de *software*, simplesmente não podem ficar satisfeitos em apenas fazê-los funcionar, necessita-se de um padrão de conceitos de código limpo e boas práticas para um bom desenvolvimento das linhas de código, pois é preciso considerar que será necessário manter a aplicação, fazendo mudanças e correções no decorrer de sua vida. Os códigos são passados de mão em mão, e o bom entendimento entre diferentes desenvolvedores é um fator crucial. Assim, o presente trabalho tem como objetivo analisar técnicas para manter um padrão de projeto. Foram utilizadas como metodologia pesquisas bibliográficas e referências analisadas no dia a dia de trabalho. Como resultado, conclui-se mostrar a grande importância de se manter um padrão para a fácil interpretação dos códigos escritos entre os times de desenvolvimento, resultando em ganhos consideráveis a favor da empresa mantenedora em termos de eficiência, tempo, agilidade e faturamento.

Palavras-chave: Código limpo. Padrão de projeto. Boas práticas.

ABSTRACT

With the large number of codes written daily by software developers, it is simply not feasible to just be satisfied to make them work, a pattern of clean code concepts and good practices is needed for a good development of the lines of code, since you must consider that you will need to keep the application, making changes and corrections in the course of its life. The codes are passed from hand to hand, and good understanding between different developers is a crucial factor. The present work aims to analyze all the techniques to maintain a design pattern. Bibliographical research and references analyzed in the daily work were used as methodology. As a result, we conclude to show the great importance of maintaining a standard for easy interpretation of written codes between development teams, resulting in considerable gains in favor of the company in terms of efficiency, time, agility and billing.

Keywords: Clean code. Project patterns. Good practices.

1 INTRODUÇÃO

Uma das principais dificuldades dos programadores certamente é a necessidade de seguir um padrão e manter um código legível e organizado. Muitos acabam escrevendo sem se preocupar com a qualidade, e sim com a velocidade a se fazer tal tarefa, pois nem sempre terão um prazo longo para a entrega.

Programadores trabalham em times, e no começo de um novo projeto, tudo flui perfeitamente, dando vida a novos módulos semanalmente, porém, nunca sempre escritos de maneira organizada e limpa. Com o passar do tempo, a velocidade das entregas vai aumentando consideravelmente pelo fato da necessidade de novas funcionalidades, contudo, não sobrando tempo para mudanças do que já foi feito. Neste mesmo ritmo, novos problemas surgirão, e pelo fato de o código poder estar mal escrito, torna-se mais caro, custoso e difícil dar as devidas manutenções (POLONI, 2018).

No entanto, os códigos são ideias escritas de forma sequencial, como se fossem histórias, portanto devem ser escritos para humanos entenderem da forma mais clara possível.

Assim, o presente artigo tem como objetivo mostrar qual a maneira correta de se criar um código limpo, instruindo o leitor a ter noção da grande importância de se ter um código bem estruturado, organizado e de fácil leitura. Primeiramente mostrando o que é um código limpo e alguns princípios de boas práticas resultando no porquê o seu uso é de extrema importância.

Com objetivo de mostrar os principais benefícios do uso de código limpo, foi utilizado como metodologia pesquisa em livros, artigos e blogs referenciados neste artigo.

2 PROCEDIMENTOS METODOLÓGICOS

A metodologia de desenvolvimento utilizada neste artigo foi a pesquisa bibliográfica, por meio de artigos científicos, blogs e livros.

Gil (2002) define que pesquisa é o procedimento racional e sistemático que tem como objetivo obter respostas a problemas propostos, sendo requerida quando não se dispõe de informações suficientes para solução dos problemas.

“Toda pesquisa o levantamento de dados de variadas fontes, quaisquer que sejam os métodos ou técnicas empregadas. Os dois processos pelos quais se podem obter os dados são a documentação direta e a indireta.” (LAKATOS; MARCONI, 1992).

3 FUNDAMENTAÇÃO TEÓRICA

3.1 Algoritmos e linguagem de programação

De acordo com Farrer et al. (1999), o computador tem como principal objetivo a resolução de problemas com grande rapidez, em questões de milissegundos, o que difere do humano, que poderia levar horas e até dias para resolver um problema matemático complexo. Um computador não consegue pensar sozinho, precisando de uma pessoa para dar os devidos comandos que resultarão em ações para a resolução de problemas, que no caso, seriam os algoritmos (sequência de passos ordenados e finitos).

Computadores só podem executar diretamente algoritmos em linguagem de baixo nível (linguagem que se aproxima mais da linguagem de máquina e se distancia mais da nossa língua), algo difícil de ser utilizado pelos seres humanos, então surgiu a ideia de linguagem de alto nível (linguagem cuja sintaxe se aproxima mais da nossa língua e se distanciam mais da linguagem de máquina), que após ser escrito algum algoritmo, com o uso de um compilador de código, ele transforma esse código em linguagem de baixo nível novamente, para que o computador possa interpretar e fazer os devidos processamentos.(FARRER et al., 1999)

São inúmeras as linguagens de programação com as quais se consegue implementar um algoritmo. Como exemplos: Java, C++, Ruby, Python, JavaScript, C#, PHP, entre outras. O conjunto de palavras escritas a partir de uma linguagem de programação constituem o código fonte da aplicação, que será o foco do presente trabalho. Este código fonte pode ser escrito fazendo o uso de boas práticas, visando facilitar seu entendimento e o princípio de código limpo pode auxiliar esta tarefa, conforme descreve o item a seguir.

3.2 Princípio de código limpo

Para Roberto (2018), um código limpo nada mais é que a aplicação de uma simples técnica para tornar fácil a compreensão de um código. A filosofia do código limpo tem como principal objetivo tornar as tarefas de leitura e escrita de códigos mais fáceis e intuitivas, mostrando sua real intenção.

Os programadores são considerados como autores, criando seus próprios códigos em linhas de raciocínios diferenciadas entre todos. Segundo Martin (2009), os códigos não são

apenas palavras jogadas em um editor de código fonte, eles precisam ser organizados de tal forma, que outros programadores consigam ler e entender facilmente. Muitas vezes são baseados em outros códigos para construir o próprio.

Conforme Martin (2009), existe um termo chamado, “A regra de escoteiro”, que é usado na maior organização de jovens escoteiros dos EUA (*A Boys Scouts of America*) que condiz: “Deixe a área do acampamento mais limpa do que como você a encontrou”. Esse pensamento voltado para os códigos seria, deixar o código mais limpo do que estava antes de alterá-lo, ou seja. manter o código sempre limpo, como, por exemplo, trocar nomes de variáveis para que fiquem claras condizendo com o motivo pelo qual foram declaradas, dividir funções extensas em blocos menores e eliminar repetições de código, entre outros.

Dessa forma, seguir algumas boas práticas são realmente necessários, tais como:

- **Definição de nomes:** nomear variáveis, funções, parâmetros, classes e métodos, é de grande importância, portanto deve-se definir um nome adequado para se ter um bom entendimento. Zanette (2017) diz que ao definir um nome existe a necessidade de ser preciso, sem dar voltas, sendo conciso e direto e não ter medo de nomes grandes.

- **Comentários:** Martin (2009) diz que o uso de comentários é um mal necessário quando não é possível expressar-se apenas em código, ou seja, quando códigos ruins são criados. Antes de escrever um comentário, averiguar se não há uma forma de melhorar o código. Comentários nem sempre dizem a verdade, eles acabam ficando desatualizados com o tempo. Porém, certos comentários são bons e devem ser usados, como, por exemplo, explicar como funciona um método de formatação de CPF e CNPJ, mostrado na Figura 1.

Figura 1 – Formatação de CPF e CNPJ

```
public static class FormatarCnpjCpf
{
    /// <summary>
    /// Formatar uma string CNPJ
    /// </summary>
    /// <param name="CNPJ">string CNPJ sem formatacao</param>
    /// <returns>string CNPJ formatada</returns>
    /// <example>Recebe '99999999999999' Devolve '99.999.999/9999-99'</example>

    public static string FormatarCNPJ(string CNPJ)
    {
        return Convert.ToUInt64(CNPJ).ToString(@"00\.000\.000\/0000\-00");
    }

    /// <summary>
    /// Formatar uma string CPF
    /// </summary>
    /// <param name="CPF">string CPF sem formatacao</param>
    /// <returns>string CPF formatada</returns>
    /// <example>Recebe '99999999999' Devolve '999.999.999-99'</example>

    public static string FormatarCPF(string CPF)
    {
        return Convert.ToUInt64(CPF).ToString(@"000\.000\.000\-00");
    }
```

Fonte: Azevedo (2016), adaptado pelos autores.

- **Funções:** assim como nos métodos, as funções levam a mesma linha de raciocínio, devem ser curtas, objetivas e simples, ou seja, cada função deve ter o seu papel de fazer uma única coisa.

- **Repetição de Código:** cada classe, método, função do sistema deve possuir apenas uma lógica, evitando códigos duplicados. Quando existe duplicação, a manutenção se torna um problema, que sua consequência é a alteração de todas as partes do código que foram copiadas, assim resultando em perca de tempo. Para uma solução eficaz, basta abstrair códigos repetidos em métodos.

- **Formatação:** a boa prática de indentar (formatar) facilita a leitura do código, como não deixar linhas de códigos maiores que a tela (linhas de códigos com muitas palavras que ultrapassam o limite do visor), e classes muito grande (LIMA, 2018). Martin (2009), afirma que "a formatação serve como uma comunicação, e essa é a primeira regra nos negócios de um desenvolvedor profissional."

- **Refatoração:** nada mais é do que reescrever códigos, porém sem alterar seu comportamento, apenas melhorando seu conteúdo. Códigos que apenas funcionam são bem diferentes de códigos feitos com qualidade, portanto a solução é refatorá-los.

Martin (2009) ainda ressalta que códigos ruins, que apenas "funcionam", são criados rapidamente pela necessidade do momento, e a longo prazo, essa má prática se torna comum, tornando esse código totalmente "sujo" e difícil de ler. Qualquer alteração realizada poderá causar falhas em outras partes da aplicação, aumentando ainda mais a dificuldade de entendimento e como consequência observa-se uma diminuição da produtividade. Assim, quando esse código é encaminhado para outros desenvolvedores, levariam horas, até dias para interpretá-lo e, a única solução, seria refatorá-lo.

Além dos princípios de código limpo observados anteriormente, existem outras boas práticas que podem ser aplicadas aos códigos para torná-los cada vez mais legíveis, e algumas dessas boas práticas são focadas em linguagens orientadas a objetos, como, por exemplo, os princípios de SOLID aliados a um projeto ágil, descritos nos itens subsequentes.

3.3 Projeto ágil

Martin e Martin (2011) dizem que em uma equipe ágil a visão global evolui com o software e não perde tempo com requisitos e funcionalidades futuras, preocupando-se com o sistema atual para torná-lo o melhor possível. Os sintomas de um projeto ruim são identificados pela violação de um ou mais princípios de projeto, como, por exemplo, os princípios SOLID (resultado de décadas de experiencia em engenharia de software), que ajudam os desenvolvedores a eliminar conhecidos problemas de desenvolvimento, visando a criação de melhores projetos para um determinado conjunto de funcionalidades. A seguir, podem ser observados alguns conhecidos problemas que permeiam a estrutura global de um software:

• **Rigidez:** tendência de software difícil de ser alterado. Quando modificado um único trecho do código, são necessárias alterações subsequentes em módulos dependentes que, por consequência, exige mais esforço do que o estimado.

• **Fragilidade:** tendência de um projeto quebrar em muitos lugares quando uma única alteração é feita. Famosos módulos que quanto mais são alterados, pior ficam e estão sempre visíveis nos catálogos de erros.

• **Imobilidade:** projetos que contêm partes que podem ser usadas em outras aplicações, porém demandam um grande esforço e projetam um elevado risco na separação dessas partes.

• **Viscosidade:** projeto cujo código é difícil de se preservar, consequentemente desenvolvedores optam em ajustar de uma maneira rápida quase sempre sem eficiência.

• **Complexidade desnecessária:** projetos que contém elementos que não serão uteis no momento, com funcionalidades que antecipam mudanças futuras, gerando um código ilegível, pesado e complexo de se entender.

• **Repetição desnecessária:** quando o mesmo código aparece inúmeras vezes, evidenciando uma falta de abstração. Códigos redundantes, quando necessitam de alterações, provocam um árduo trabalho, pois cada repetição contida no projeto deve ser alterada.

• **Opacidade:** refere-se à dificuldade de compreensão de um módulo. Desenvolvedores devem pensar em refatorar sempre o código de tal maneira que outros possam ler e entender sem problemas, para evitar futuros retrabalhos.

3.4 Princípios S.O.L.I.D.

Segundo Campomori (2017), a sigla S.O.L.I.D. refere-se a princípios que visam a correta aplicação dos conceitos de orientação a objetos, mantendo a arquitetura do *software* correta e desacoplada.

Criado por Michael Fathers, S.O.L.I.D. representa um acrônimo de cinco princípios da programação orientada a objetos, teorizados por Robert C. Martins (Uncle Bob) por volta do ano 2000 (AZEVEDO, 2018). Os cinco princípios seriam:

- **SRP - *Single responsibility principle* (Princípio da Responsabilidade Única)**

Classes ou módulos devem ter apenas uma única responsabilidade, ressaltando a coesão do código. De acordo com Aniche (2015), “classe coesa é aquela que possui uma única responsabilidade. Ou seja, ela não toma conta de mais de um conceito no sistema. Se a classe é responsável por representar uma Nota Fiscal, ela representa apenas isso. As responsabilidades de uma Fatura, por exemplo, estarão em outra classe.”

Classes com muitos métodos, modificadas frequentemente nunca parando de crescer, devem ser analisadas e refatoradas para respeitar o princípio SRP e ter responsabilidade única.

- **OCP - *Open/closed principle* (Princípio do Aberto/Fechado)**

Grande parte dos investimentos está relacionado a manutenção de código. Quando um código é mal feito, sem padrões adequados e boas práticas, qualquer alteração mínima pode acarretar problemas em outras partes do sistema. O padrão *Open Close Principle*, um dos princípios SOLID ressalta: **“**Entidades de *software* (classes, métodos, módulo, etc.) devem estar abertas para extensão, mas fechadas para modificação**”.**

O módulo ou classe pode ser ampliado com novas funcionalidade, estendendo facilmente o seu comportamento, porém devem ser fechados para modificações em seu comportamento padrão.

Martin e Martin (2011) cita que "um módulo pode manipular uma abstração. Tal módulo pode ser fechado para modificação, pois ele depende de uma abstração fixa. Apesar disso, o comportamento desse módulo pode ser ampliado pela criação de novas derivadas da abstração."

- **LSP - Liskov *substitution principle* (Princípio da Substituição de Liskov)**

Em 1988, Barbara Liskov escreveu que “O que se deseja aqui é algo como a seguinte propriedade de substituição: se para cada objeto o1 do tipo S existe um objeto o2 do tipo T, tal que, para todos os programas P definidos em termos de T, o comportamento de P fica inalterado quando o1 é substituído por o2, então S é um subtipo de T.”

Em outras palavras, classes derivadas podem ser usadas no lugar das classes bases e classes bases podem ser usadas por qualquer umas de suas subclasses, mas evidenciando-se o cuidado com a utilização de heranças, validando se o polimorfismo realmente faz sentido (FELIPE, 2018).

- **ISP - *Interface segregation principle* (Princípio da Segregação de Interfaces)**

Para Aniche (2015), interfaces são como classes e módulos e também devem ser coesas, possuindo uma única responsabilidade, possibilitando um maior reuso e evitando improvisos e soluções paliativas.

Já Martin e Martin (2011) afirmam que “o ISP reconhece que existem objetos que exigem interfaces não coesas; contudo, sugerem que os clientes não devem reconhece-las como uma única classe. Em vez disso, os clientes devem reconhecer classes base abstratas que tem interfaces coesas.”

- **DIP - *Dependency inversion principle* (Princípio da Inversão de Dependências)**

O conceito diz que “Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações; Abstrações não devem depender de detalhes; Detalhes devem depender de abstrações”.

Em outras palavras, dependa de uma abstração (interface) e não de uma implementação (classe abstrata). Abstrair não significa utilizar classes abstratas, e sim não implementar fazendo que funcione sem ter que criar um acoplamento, visando não utilizar dependência (herança) direta entre classes (NOGUEIRA, 2018).

3.5 Padrões de projeto

O trabalho realizado utilizando alguns padrões pode gerar benefícios, como, por exemplo, economia de tempo e energia. Beck (2013) descreve que usar padrões ajudam programadores a escreverem soluções simples para problemas comuns, deixando mais tempo, energia e criatividade para serem aplicados em problemas verdadeiramente únicos e específicos. Caso a equipe esteja insatisfeita com um padrão, pode-se haver uma discussão para o uso de um novo padrão.

Beck (2013) ainda afirma que "copiar cegamente o estilo de alguém não é tão eficaz quanto pensar a respeito e praticar seu próprio estilo, além de discutir e compartilhar um estilo com sua equipe".

Após o entendimento das boas práticas de programação utilizando os princípios do código limpo e de SOLID junto aos padrões de projeto, torna-se possível realizar uma discussão sobre o assunto em meio ao ambiente de desenvolvimento de *software* como pode ser observado no item a seguir.

4 DISCUSSÃO

A grande demanda na construção e manutenção de sistemas vem aumentando rapidamente com o passar dos anos. Códigos são escritos diariamente, possivelmente, sem uma organização ou padrão para ser seguido e escrito da maneira correta. Muitos programadores desenvolvem códigos mal feitos, por falta de tempo, orientação e conhecimento, e posteriormente, nem mesmo os próprios autores conseguem interpretar seus códigos.

Códigos mal feitos, escritos sem as devidas atenções, apenas focando em fazê-los funcionar o mais rápido possível podem gerar problemas futuros complicados de serem corrigidos que certamente irão elevar o tempo de manutenção, ainda mais quando a pessoa responsável pelos ajustes não é a mesma pessoa que criou esse código.

O tempo gasto atualmente tentando interpretar o que códigos mal escritos estão fazendo é alto, resultando em atrasos nas entregas e possíveis multas para as empresas mantenedoras.

Ser consciente e fazer uso dos conceitos de código limpo, padrões de código e boas práticas de programação é de extrema importância para a escrita correta de funcionalidades

através da arte da codificação. O tempo utilizado para elaborar o nome de uma variável, se atentar em manter métodos objetivos e curtos, utilizar corretamente os princípios do SOLID, terá como consequência o ganho de tempo futuro, pois será mais fácil entender e modificar o código desenvolvido

5 CONSIDERAÇÕES FINAIS

Desenvolvedores de *software* são autores de seus códigos assim como escritores são autores de seus livros, sendo exclusivamente responsabilidade deles manter um código de boa qualidade.

Utilizar os conceitos citados neste trabalho, ajudará o próprio autor do código e até mesmo outros desenvolvedores entenderem da forma mais simples o seu propósito. Cabe a cada um ter um bom senso de se usar as boas práticas e desenvolver aplicações de qualidade. Desta forma, ressalta-se a importância de apresentar estes conceitos para desenvolvedores que ainda não os conhecem, destacando os verdadeiros motivos de se escrever códigos de boa qualidade.

REFERÊNCIAS

ANICHE, Maurício. **Orientação a Objetos e SOLID para Ninjas: Projetando classes flexíveis.** São Paulo: Casa do Código, 2015. *E-book.*

AZEVEDO, Antonio Carlos Ferreira. **Formatar CNPJ e CPF em C# (CSharp)**. [*S. l.*], 26 fev. 2016. Disponível em: http://www.codigoexpresso.com.br/Home/Postagem/28. Acesso em: 13 abr. 2019.

AZEVEDO, Mariana. **Princípios S.O.L.I.D.: o que são e porque projetos devem utilizá-los.** [*S. l.*], 4 ago. 2018. Disponível em: <https://medium.com/equals-lab/princ%C3%ADpios-s-o-l-i-d-o-que-s%C3%A3o-e-porque-projetos-devem-utiliz%C3%A1-los-bf496b82b299>. Acesso em: 2 abr. 2019.

BECK, Kent. **Padrões de implementação: Um catálogo de padrões indispensável para o dia a dia do programador.** Porto Alegre: Bookman, 2013.

CAMPOMORI, Cleber. **Introdução aos Princípios SOLID.** [*S. l.*], 9 jul. 2017. Disponível em: https://www.treinaweb.com.br/blog/introducao-aos-principios-solid/. Acesso em: 2 abr. 2019.

FARRER, Harry *et al*. **Algoritmos Estruturados.** Minas Gerais: Livros Técnicos e Científicos Editora S.A., 1999.

FELIPE, Marcos. **Princípios SOLID: O Princípio da Substituição de Liskov**. [*S. l.*], 31 jan. 2018. Disponível em: http://dtidigital.com.br/blog/principios-solid-o-principio-da-substituicao-de-liskov/. Acesso em: 13 abr. 2019.

GIL, Antonio Carlos. **Como Elaborar Projetos de Pesquisa**. São Paulo: Atlas S.A., 2002.

LAKATOS, Eva Maria; MARCONI, Marina de Andrade. **Metodologia do Trabalho Científico**. São Paulo: Atlas S.A., 1992.

LIMA, Victor. **As 4 regras do código limpo**. [*S. l.*], 9 out. 2018. Disponível em: https://blog.schoolofnet.com/as-4-regras-do-codigo-limpo/. Acesso em: 13 abr. 2019.

MARTIN, Robert C. **Código Limpo: Habilidades Práticas do Agile Software.** [*S. l.*]: Alta Books, 2009.

MARTIN, Robert C.; MARTIN, Micah. **Princípios, Padrões e Práticas Ágeis em C#.** São Paulo: Bookman, 2011. PDF.

NOGUEIRA, Wagner. **Princípio da Inversão de Dependência—DIP**. [*S. l.*], 21 ago. 2018. Disponível em: https://medium.com/@engnogueirawgn/princ%C3%ADpio-da-invers%C3%A3o-de-depend%C3%AAncia-dip-2a04d83f7b9e. Acesso em: 13 abr. 2019.

POLONI, J. **Código Limpo: Práticas para contar uma história com seu código**. [*S. l.*], 2 mar. 2018. Disponível em: http://bluedev.com.br/2018/03/02/codigo-limpo-praticas-para-contar-uma-historia-com-seu-codigo/. Acesso em: 13 abr. 2019.

ROBERTO, João. **Clean Code: O que é? Por que usar?** [S. l.], 7 ago. 2018. Disponível em: <https://medium.com/joaorobertopb/1-clean-code-o-que-%C3%A9-porque-usar-1e4f9f4454c6>. Acesso em: 28 mar. 2019.

ZANETTE, Alysson**. Introdução aos Princípios SOLID.** [*S. l.*], 10 abr. 2017. Disponível em: https://becode.com.br/clean-code/. Acesso em: 2 abr. 2019.