# O código limpo e suas diretrizes

Em poucas palavras, código limpo significa que o código é fácil de ler, fácil de entender e fácil de mudar (TAKAGI, 2021). É tão simples quanto ter variáveis, funções e classes descritivamente nomeadas para que, ao ler, saiba o que cada um contém ou faz sem adivinhação (HAZIN, et. al., 2017).

Martim (2019), menciona sobre o emprego desses princípios-chave: KISS, que significa *Keep It Simple, Stupid, e DRY, Don't Repeat Yourself.* Pergunte a si mesmo se há uma solução melhor para resolver problemas ou complexidade dentro do seu código. Comece a construir hábitos que sejam propícios para limpar códigos, tais como:

* Verificar suas convenções de nomeação. Mantenha-os limpos e conciso;
* Ser consistente. Usar os mesmos nomes em funções semelhantes (*Get vs Fetch, Set vs Update, Add vs Append*);
* Se copiar o código várias vezes, considere melhores maneiras de tornar seu código mais eficiente;
* Para solicitações de banco de dados, mantenha-o o mais simples e produtivo possível.

Se achar que um pedaço de código é muito complexo ou não é a melhor maneira de fazê-lo, então pare. Dê a si mesmo alguns momentos para repensar o processo e ver se pode chegar a uma maneira melhor. Realizar uma revisão com colegas, pode ser uma boa opção (MANGRICH, et. al., 2021). Sempre tenha em mente a importância de se comparar o código de sua equipe com o seu e esteja aberto a sugestões, pois assim a empresa sempre trabalhará focada e com códigos mais robustos.

A maioria dos códigos não é escrito e esquecido, é escrito e depois modificado ao longo do tempo para mantê-lo atualizado e eficiente no que diz respeito ao projeto (TAKAGI, 2021). Provavelmente, não é a única pessoa que precisará entender o que seu código faz, e é por isso que o que ele faz precisará ser óbvio. É possível usar “comentários” no código?

Sim, mas se essa é sua única forma de comunicação dentro do seu código, então deve-se estar escrevendo código sujo. Ter certeza de que seu código pode falar por si mesmo também irá ajudá-lo a desenvolver seu vocabulário técnico. Isso beneficia elevando sua qualidade de código, quando se pode discutir o código claramente, pode-se pedir ajuda e sugestões mais eficazes sobre como melhorar o que se escreve. Código limpo não se escreve sozinho, é preciso um foco dedicado em apresentar o que se quer transmitir (MARTIN, 2019).

Mas o porquê é importante considerar estas diretrizes? (HAZIN, et. al., 2017), explica que uma equipe não estará no campo de desenvolvimento de software por muito tempo antes de se juntar a outro projeto que é uma colaboração em tempo real ou contém código legado. Trabalhar com código que os outros têm ou trabalharão torna evidente que o código limpo é importante.

Saltar em código que não tem uma direção clara é demorado para todas as partes envolvidas, porque quando não é aparente o que o código faz ou por que ele faz, se desperdiça recursos tentando descobrir seu propósito (MANGRICH, et. al., 2021). É reutilizável? Quanto mais limpo for o código, maior a probabilidade de se reutilizá-lo! Escrever código limpo faz de um programador, mais eficiente porque está escrevendo código que faz uma coisa clara, que não leva tempo extra para escrever. O código limpo também torna a manutenção mais fácil e rápida (CARVALHAES, et. al., 2021).

Pedir ajuda de um colega e depois ter que percorrer um código mal escrito para encontrar o problema nunca é um impulso para sua confiança. Escrever código limpo permitirá que se produza código de qualidade que se orgulha de compartilhar com a equipe, sabendo que eles terão mais facilidade em trabalhar nele também (MARTIN, 2019).

Normalmente, nem sempre se escreve um código limpo (MANGRICH, et. al., 2021). E preciso ter a rotina da escrita, “treinar” junto a equipe para levar a prática da perfeição. Quando se sabe melhor, pode-se fazer melhor. Se tirar um tempo para escrever código limpo agora, colherá os benefícios da legibilidade e da depuração mais fácil para frente.

Se achar que passa a maior parte do tempo escrevendo código, está errado. (CARVALHAES, et. al., 2021), menciona que se gasta mais tempo lendo códigos, descobrindo bugs, encontrando problemas e descobrindo como resolvê-los. Código limpo torna todas essas coisas mais fáceis. Se há uma coisa em que todos os programadores concordam, é que código limpo é melhor (MANGRICH, et. al., 2021).

# O ciclo de desenvolvimento orientado a testes funcionais (TDD)

O conceito de Desenvolvimento Orientado por Testes (TDD) foi introduzido em 2003 por Kent Beck (WAGNER; FANTONI, 2021). Não há definição formal, mas Beck dá abordagens e exemplos de TDD. O objetivo do TDD é "escrever código limpo que funcione" (HAZIN, et. al., 2017).

No TDD, siga apenas uma regra de ouro: Só altere o código de produção se algum teste falhar. Caso contrário, apenas refator para otimizar o código. Para os requisitos atualizados, converta-os em casos de teste, adicione esses testes e, só então, escreva um novo código (MEIRELES; DE RESENDE COSTA, 2021).

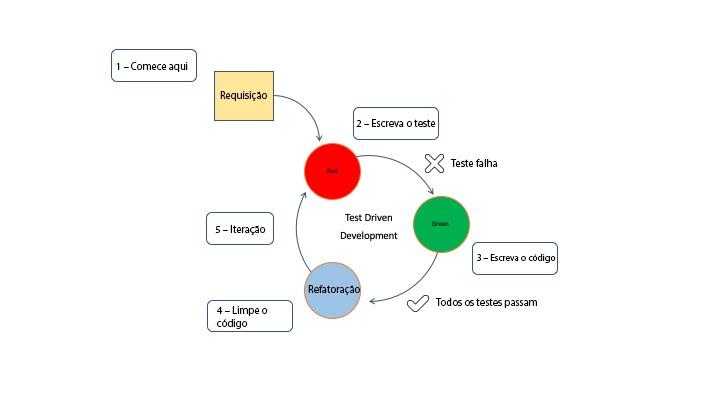
TDD é um ciclo de desenvolvimento muito curto e repetitivo. Os requisitos do cliente são transformados em casos de teste altamente específicos e o software é escrito e melhorado para passar nos novos testes (WAGNER; FANTONI, 2021).

O Desenvolvimento Orientado por Testes está relacionado a conceitos de programação em programação extrema, defendendo atualizações/lançamentos de software frequentes em ciclos curtos de desenvolvimento e promovendo extensas revisões de código, testes de unidade e adição incremental de recursos (MEIRELES; DE RESENDE COSTA, 2021).

Um conceito intimamente relacionado ao TDD é o AtDD (*Acceptance Test-Driven Development*, desenvolvimento orientado para testes de aceitação), onde o cliente, o desenvolvedor e o testador participam do processo de análise de requisitos (HAZIN, et. al., 2017). O TDD é tanto para desenvolvedores de aplicativos móveis quanto para aplicativos web, enquanto o ATDD é uma ferramenta de comunicação para garantir que os requisitos sejam bem definidos (WAGNER; FANTONI, 2021).

De Matos et. al., (2018), menciona que o ciclo TDD, também conhecido como processo Vermelho-Verde-Refator, pode ser observado na Figura 1 abaixo, demonstra que:

Figura 1 - Ciclo do desenvolvimento orientado para o teste – TDD



Fonte: (De Matos et. al., 2018) – adaptado pelo autor

1. Adicione um teste, que certamente falhará. (Vermelho): No TDD, cada recurso em um software é adicionado pela primeira vez em termos de casos de teste. Um teste é criado para uma função nova ou atualizada. Para escrever os testes, os desenvolvedores devem entender as especificações e requisitos do recurso. Essa prática separa o TDD dos métodos tradicionais de desenvolvimento de software, onde os testes de unidade são escritos após a escrita do código-fonte. Desta forma, o TDD faz com que o desenvolvedor se concentre nos requisitos antes de escrever o código.

2. Execute todos os testes. Veja se algum teste falha: A execução de testes valida que o arreio de teste está funcionando corretamente e simultaneamente prova que, à medida que novos testes adicionados estão falhando com o código existente, um novo código é necessário.

3. Escreva apenas código suficiente para passar em todos os testes. (Verde): O novo código escrito nesta fase pode não ser perfeito e pode passar no teste de forma irrelevante. O único requisito nesta fase é que todos os testes sejam aprovados. Uma maneira possível de começar adicionando as declarações é retornar uma constante e adicionar gradualmente blocos lógicos para construir a função.

4. Execute todos os testes. Se algum teste falhar, volte para o passo 3. Caso contrário, continue: Se todos os testes forem aprovados, pode-se dizer que o código atende aos requisitos de teste e não degrada quaisquer recursos existentes. Se algum teste falhar, o código deve ser editado para garantir que todos os testes passem.

5. Refatorar o código. (Refator): À medida que a base de código cresce, ela deve ser limpa e mantida regularmente. Como? Há algumas maneiras:

* Um novo código que poderia ter sido adicionado por conveniência para passar em um teste pode ser movido para seu lugar lógico no código.
* A duplicação deve ser eliminada.
* Definições de objeto e nomes devem ser definidos para representar seu propósito e uso.
* À medida que mais recursos são adicionados, as funções se tornam longas. Pode ser benéfico dividir e nomear cuidadosamente para melhorar a legibilidade e a manutenção.
* Como todos os testes são reexecutados ao longo da fase de refatoração, o desenvolvedor pode estar confiante de que o processo não altera nenhuma funcionalidade existente.

6. Se um novo teste for adicionado, repita a partir do passo 1. Dê pequenos passos, mirando apenas 1 a 10 edições entre cada teste executado.

Se o novo código não satisfaz rapidamente um novo teste, ou outros testes não relacionados falharem inesperadamente, então desfaça/reverta para um código de trabalho, em vez de fazer depuração extensiva.

Ao usar bibliotecas externas, é importante não fazer incrementos tão pequenos que apenas testem a biblioteca em si, a menos que seja para testar se a biblioteca está desatualizada/incompatível, *buggy* ou não completa de recursos.

# Programação orientada a testes funcionais na prática

De Mattos et. al., (2018), menciona que para sistemas de grande porte, o teste funcional é desafiador e requer uma arquitetura modular com componentes bem definidos. Alguns requisitos-chave que devem ser cumpridos são:

* A alta coesão garante que cada módulo forneça um conjunto de capacidades relacionadas, facilitando a manutenção dos testes correspondentes.
* O acoplamento baixo permite testes isolados de módulos.

Na Modelagem de Cenários, um conjunto de gráficos de sequência é construído, cada gráfico focado em um único cenário de execução em nível de sistema. Ele fornece um excelente veículo para criar estratégias de interação em resposta a uma entrada. Cada Modelo de Cenário serve como um conjunto de requisitos para os recursos que um componente fornecerá. A modelagem de cenários pode ser útil na construção de testes TDD em sistemas complexos (SARCINELLI, 2018).

É importante diferenciar o código entre teste funcional e a produção (DE SOUZA, JHONATA LIMA, 2022). O conjunto de testes funcionais da unidade deve ser capaz de acessar o código para testar. No entanto, o desenho de critérios como ocultação de informações e encapsulamento e separação de módulos não deve ser comprometido.

No design orientado para objetos, os testes funcionais ainda não poderão acessar membros e métodos de dados privados e exigir uma codificação extra. Alternativamente, uma classe interna pode ser usada dentro do código fonte para conter os testes da unidade. Tais hacks de teste não devem permanecer no código de produção. Os praticantes de TDD frequentemente argumentam se os dados privados devem mesmo ser testados (DE MATTOS, et. al., 2018).

Barros et. al., (2019), cita que com base em linguagens de programação únicas, existem várias estruturas que suportam o desenvolvimento orientado a testes, como:

1. **csUnit e NUnit** – Ambos são estruturas de teste de unidades de código aberto para projetos .NET.
2. **PyUnit e DocTest:** Estrutura de teste da Unidade Popular para Python.
3. **Junit:** Ferramenta de teste de unidade amplamente utilizada para Java
4. **TestNG:** Outra popular estrutura de testes Java. Este quadro supera as limitações do Junit.
5. **Rspec:** Uma estrutura de testes para projetos Ruby

O processo de entrega de produtos de qualidade exige não apenas depuração, mas também exige otimização no processo de desenvolvimento. Quando incorporada corretamente, a abordagem TDD proporciona inúmeros benefícios, particularmente em termos de trazer custo-benefício no longo prazo e fornecer verdadeiro valor para as empresas (BARROS, et. al., 2019).

# Programação Orientada a Tipos: conceitos, diretrizes e importância para as empresas desenvolvedoras de softwares

A programação orientada a tipos é um paradigma que abrange abordagens funcionais e orientadas a objetos para escrever código tipado estaticamente (SILVA, et al., 2019). A maioria das linguagens de programação nesta categoria faz uso extensivo de subtipagem e polimorfismo paramétrico. Esta seção introduz os conceitos básicos, a maioria deles conhecidos de linguagens funcionais e orientadas a objetos, e depois passa a explicar algumas técnicas mais envolvidas usadas em linguagens com poderosos sistemas de tipos.

Muito trabalho tem sido investigado com paradigmas de programação (CLEMENTE, 2021). Paradigmas comuns incluem imperativo, funcional, orientado a objetos e orientado a aspectos. No entanto, no desenvolvimento de ideias de programação orientada a tipos, toma-se conceito familiar de um tipo, associa-se em profundidade a semântica de tempo de execução a tal, de modo que o comportamento do uso da variável (ou seja, acesso e atribuição) pode ser determinado pela análise do tipo específico (ROCHA; NETO, 2003).

Em muitas linguagens existe a necessidade de combinar um número de atributos com uma variável, para este fim permitimos que o programador combine tipos juntos para formar um supertipo (cadeia de tipos).

Uma cadeia de tipos é uma coleção de tipos, combinados pelo programador (VEDOVATTO; COSTA, 2013). É essa cadeia de tipo que determinará o comportamento de uma variável específica. A precedência na cadeia de tipos é da direita para a esquerda (ou seja, o último tipo adicionado substituirá o comportamento dos tipos adicionados anteriormente). Essa precedência permite que o programador adicione informações adicionais, seja de forma terminante ou para uma expressão específica, à medida que o código progride (ANDRADE, et al., 2004).

As variáveis de tipo são um conceito interessante. Semelhante às variáveis normais do programa, elas são declaradas para conter uma cadeia de tipos. Ao longo da execução do programa, elas podem ser tratadas como variáveis normais do programa e podem ser verificadas por meio de condicionais, atribuídas e modificadas (SILVA, et al., 2019).

Rocha e Neto (2003), mencionam que há uma série de vantagens na programação orientada a tipos, incluindo:

* Eficiência - A rica quantidade de informações permite que o compilador execute muita análise estática e otimização, resultando em maior eficiência.
* Simplicidade - Ao fornecer uma biblioteca de tipos limpa, o programador pode usar tipos bem documentados para controlar muitos aspectos de seu código.
* Linguagem mais simples - Ao tirar a maior parte da complexidade da linguagem e colocá-la em uma biblioteca de tipos fracamente acoplada, a linguagem é mais simples do ponto de vista do design e da implementação (do compilador). Adicionar inúmeras palavras-chave de linguagem geralmente resulta em um design frágil, usando programação orientada a tipos, isso é evitado
* Facilidade de manutenção - Ao alterar o tipo, pode-se ter um efeito considerável na semântica do código, abstraindo o programador isso torna o código mais simples, mais flexível e mais fácil de manter.

Todas as linguagens paralelas atuais sofrem com o comprometimento da simplicidade versus eficiência. Ao abstrair o programador longe dos detalhes de baixo nível, dá-lhes uma linguagem simples de usar, mas o alto nível de informações fornecidas ao compilador permite que muita análise seja realizada durante a fase de compilação. A partir de linguagens de baixo nível (como C), é difícil para o compilador entender como o programador está usando o paralelismo, portanto, a otimização de tal código é limitada (ROCHA; NETO, 2003).

Fornece-se ao programador a escolha entre programação explícita e implícita - eles podem confiar nos padrões de linguagem embutidos e seguros ou, alternativamente, usar tipos adicionais para obter mais controle (e desempenho). Portanto, a linguagem é aceitável tanto para o programador paralelo novato quanto para o especialista (CLEMENTE, 2021).

Vedovatto e Costa (2013), mencionam que a programação orientada a tipos pode ser usada em:

* Programação GUI - A programação GUI pode ser bastante cansativa e repetitiva (daí o uso de IDEs de design gráfico). Usando tipos, isso abstrairia o programador de muitas das questões repetitivas.
* Retrofit Existing Languages - A abordagem de tipo poderia ser aplicada a linguagens existentes onde um retrofit poderia ser realizado, mantendo o programador em sua zona de conforto, mas também lhes dando o poder da programação orientada a tipos.
* Numerosos sistemas de tipo - O sistema de tipos é completamente separado da linguagem real, seria possível fornecer um número de sistemas de tipo para uma única linguagem, como um sistema *paralelo*, um sistema *sequencial* etc.

Sendo assim, Maguire (2018), relata que a programação orientada a tipos é uma das abordagens mais populares e fundamentais na construção de softwares modernos. É uma técnica que permite criar estruturas de dados complexas, seguras e eficientes por meio do uso de tipos estáticos (THOMPSON, 1991). Tipos estáticos são definidos em tempo de compilação e verificam a validade das operações realizadas com os objetos de uma classe, evitando assim erros e reduzindo a complexidade do código. Nesse sentido, a programação orientada a tipos é considerada uma das melhores práticas para construir softwares confiáveis e escaláveis (SCOTT, 2000).

Uma das vantagens da programação orientada a tipos é a sua capacidade de detectar erros de tipagem em tempo de compilação (PIERCE, 2002). Isso significa que o compilador verifica se as operações realizadas em uma variável são compatíveis com o tipo declarado. Caso contrário, o compilador exibe uma mensagem de erro e impede a compilação do programa. Essa verificação estática ajuda a evitar erros de tempo de execução e facilita a detecção de erros de programação (MAGUIRE, 2018).

Abaixo, segue um pequeno exemplo de um código fonte, desenvolvido em Haskell (que será abordado na seção XX), evidenciando a estrutura de um pequeno programa orientado a tipos.

Exemplo 01: exemplo de programação orientada a tipos, tendo como o uso do tipo de dados 'Person para definir uma pessoa com nome e idade, onde:

***data Person = Person String Int***

***personName :: Person -> String***

***personName (Person name \_) = name***

***personAge :: Person -> Int***

***personAge (Person \_ age) = age***

Nesse exemplo, definimos um novo tipo de dados ***Person***, que representa uma pessoa com um nome e uma idade. Em seguida, definimos duas funções ***personName*** e ***personAge***, que acessam o nome e a idade de uma pessoa, respectivamente. Observe que o tipo de dados ***Person*** é definido usando a palavra-chave ***data***, que é usada para definir tipos de dados personalizados em Haskell. Para criar uma pessoa usando o tipo de dados ***Person***, pode-se usar o seguinte código: ***let john = Person "John Doe" 30.***

Nesse exemplo, criamos uma nova pessoa chamada john com nome "John Doe" e idade 30. Podemos então acessar o nome e a idade de john usando as funções ***personName*** e ***personAge***, como mostrado abaixo:

***print (personName john) -- Output: "John Doe"***

***print (personAge john) -- Output: 30***

Além disso, a programação orientada a tipos ajuda a reduzir a complexidade do código, tornando-o mais fácil de ler, entender e manter (THOMPSON, 1991). Isso ocorre porque a definição de tipos estáticos permite que o programador saiba exatamente quais operações podem ser realizadas com um objeto de uma determinada classe. Dessa forma, a programação orientada a tipos ajuda a evitar erros de lógica e torna o código mais fácil de depurar (PIERCE, 2002).

A programação orientada a tipos é importante para o desenvolvimento de software por várias razões. Em primeiro lugar, ela ajuda a garantir a segurança e a confiabilidade do código, permitindo que os desenvolvedores capturem erros de tempo de compilação em vez de erros de tempo de execução. Isso pode reduzir significativamente o tempo e o esforço necessários para depurar e testar o software (SCOTT, 2000).

Além disso, a programação orientada a tipos permite que os desenvolvedores criem abstrações de software mais robustas e reutilizáveis (MAGUIRE, 2018). Ao definir tipos de dados personalizados, os desenvolvedores podem encapsular dados e comportamentos relacionados em um único objeto, tornando o código mais modular e fácil de entender e manter (PIERCE, 2003).

A programação orientada a tipos também pode ajudar a melhorar a eficiência do código, permitindo que o compilador otimize o código com base nas informações de tipo (THOMPSON, 1991). Por exemplo, o compilador pode usar informações de tipo para evitar a necessidade de conversões de tipo desnecessárias ou para otimizar a alocação de memória (PIERCE, 2003).

A programação orientada a tipos é uma técnica de programação bem estabelecida e amplamente utilizada, com muitas bibliotecas e estruturas de programação disponíveis para os desenvolvedores (MAGUIRE, 2018). Isso significa que, ao usar a programação orientada a tipos, os desenvolvedores podem aproveitar a experiência e os recursos da comunidade de desenvolvimento de software para acelerar o desenvolvimento de seus próprios projetos (SCOTT, 2000).

Outra vantagem da programação orientada a tipos é a sua capacidade de suportar polimorfismo e herança. Polimorfismo é a capacidade de um objeto de uma classe se comportar como outro objeto de uma classe diferente (MENEZES, 2012). Herança é a capacidade de criar uma nova classe que é uma extensão de uma classe existente, herdando suas propriedades e métodos. Esses recursos são importantes para a reutilização de código e a criação de hierarquias de classes (COSTA, 2013).

Embora a programação orientada a tipos tenha sido criticada por ser excessivamente restritiva e burocrática, muitos programadores ainda consideram essa abordagem como essencial para a construção de softwares seguros e confiáveis (DEJEU, 2013). Na verdade, a programação orientada a tipos é tão importante que muitas linguagens de programação modernas, como Java, C#, Python e Ruby, a suportam nativamente. Essas linguagens oferecem recursos avançados de programação orientada a tipos, como interfaces, classes abstratas e polimorfismo.

Por fim, a programação orientada a tipos também pode ajudar a melhorar a comunicação e a colaboração entre os membros da equipe de desenvolvimento de software. Ao usar tipos de dados bem definidos e documentados, os desenvolvedores podem se comunicar mais claramente sobre o comportamento e os requisitos do código, facilitando a colaboração e a solução de problemas em equipe (FONSECA; SANTOS, 2016).

# Bugs: Compreendendo, caracterizando e classificando os tipos de bugs

Um bug é um problema inesperado com software ou hardware (BELLER, et al., 2018). Problemas típicos são muitas vezes o resultado de interferência externa com o desempenho do programa que não foi antecipado pelo desenvolvedor (LUNA, 2022). Pequenos bugs podem causar pequenos problemas, como telas congeladas ou mensagens de erro inexplicáveis que não afetam significativamente o uso (SILVA, et al., 2020). Os principais bugs podem não apenas afetar o software e o hardware, mas também podem ter efeitos não intencionais em dispositivos conectados ou software integrado e podem danificar arquivos de dados (BELL, et al. 2018).

(Aqui pode ser mencionando a tabela sua do Excel, relacionando alguns tipos de bugs, coloque uns 10).

A palavra bug originou-se na engenharia. O uso do termo na computação é atribuído à programadora pioneira Grace Hopper. Em 1944, Hopper era um jovem oficial da Reserva Naval que trabalhou no computador Mark I em Harvard. Hopper descreveu mais tarde um incidente em que um técnico teria puxado um bug real - uma mariposa, na verdade - entre dois relés elétricos em um computador Mark II. A Marinha teve a mariposa em exibição por muitos anos. O Smithsonian agora o tem em suas propriedades (BELLER, et. al., 2015).

Embora os bugs normalmente causem falhas irritantes no computador, seu impacto pode ser mais sério (SILVA, et al., 2020). Um artigo de 2005 da Wired sobre os 10 piores bugs de software da história relatou que os bugs causaram grandes explosões, aleijaram sondas espaciais e mataram pessoas. Por exemplo, em 1982, um sistema - supostamente implantado pela Agência Central de Inteligência - controlando o gasoduto Transiberiano causou a maior explosão não nuclear da história (WIRED, 2005).

O artigo também disse que, entre 1985 e 1987, um bug, chamado de condição de raça, em um dispositivo de radioterapia resultou na entrega de doses letais de radiação, matando cinco pessoas e ferindo outras. Em 2005, a Toyota fez um recall de 160.000 Priuses porque um bug fez com que as luzes de advertência acendessem e o motor parasse sem motivo (WIRED, 2005).

Um bug é apenas um tipo de problema que um programa pode ter (BELLER, et al., 2018). Os programas podem ser executados sem bugs e ainda ser difíceis de usar ou falhar em algum objetivo principal. Esse tipo de falha é mais difícil de testar. Um programa bem projetado desenvolvido usando um processo bem controlado resulta em menos bugs por milhares de linhas de código. É por isso que é importante incluir a usabilidade nos testes (LUNA, 2022).

Bell, et al. (2018), relata sobre diferentes tipos de bugs que causam mau funcionamento dos computadores, como:

**Aritmética**. Às vezes referidos como erros de cálculo, bugs aritméticos são erros matemáticos no código que fazem com que ele não funcione.

**Interface**. Um bug de interface ocorre quando sistemas incompatíveis estão conectados ao computador. O problema pode vir de uma peça de hardware ou software. Uma interface de programação de aplicativos pode ser um exemplo de um bug de interface.

**Lógica**. Esses erros acontecem quando a lógica do script faz com que o programa produza as informações erradas ou fique preso e não forneça nenhuma saída. Um exemplo de um erro lógico é um loop infinito onde uma sequência de código é executada continuamente.

**Sintaxe**. Esses bugs vêm de código escrito com os caracteres errados. Linguagens de programação diferentes têm sintaxes diferentes, portanto, usar a sintaxe de uma pode causar um bug em outra.

**Equipe**. Este é um bug que surge quando há falta de comunicação entre os programadores. Um exemplo é quando há diferenças entre a documentação do produto e o produto. Outro exemplo é quando os comentários descrevem incorretamente o código do programa.

Já Beller et. al., (2015), cita que outra maneira simples de categorizar bugs é da perspectiva do usuário. Esses tipos de bug incluem o seguinte:

**Visual**. Um usuário pode concluir a função escolhida, mas algo parece errado com o aplicativo. Isso pode ser um problema com o design responsivo do aplicativo.

**Funcional**. Um bug funcional significa que o programa não funciona como pretendido. Por exemplo, um usuário clica no botão Salvar, mas os dados não são salvos.

Breu et. al., (2010), cita que os bugs também podem ser classificados pelo nível de dificuldade que causam ao usuário:

* Bugs de baixo impacto têm um efeito mínimo na experiência do usuário.
* Os de alto impacto afetam algum nível de funcionalidade, mas o aplicativo ainda é utilizável.
* Bugs críticos impedem a funcionalidade principal do aplicativo.

Cendron e Annes (2014), cita outra abordagem para a classificação de bugs que é observar onde eles ocorrem:

* Bugs no nível da unidade são bugs de software simples contidos em uma unidade de código. Eles são tipicamente devido a erros de cálculo ou lógica e lidam com um pedaço de software. Eles geralmente são fáceis de consertar.
* Bugs no nível do sistema são bugs mais complexos causados por vários softwares interagindo de maneiras que causam problemas.

Silva et al., (2020), menciona que os bugs fora de limite surgem quando o usuário interage com o programa de uma maneira inesperada. Por exemplo, isso acontece quando um usuário insere um parâmetro em um campo de formulário que o programa não foi projetado para manipular. Bugs fora da conexão podem ser usados para explorar software (BREU, et. al., 2010). Por exemplo, os agentes de ameaças usam a vulnerabilidade Infra: *Halt* para realizar ataques de envenenamento de cache do sistema de nomes de domínio na tecnologia operacional.

Segundo Cendron e Annes (2014), várias maneiras de resolver bugs, dependendo do tipo de bug e onde e quando eles são encontrados. A melhor maneira de resolver erros de programação é através da prevenção (LEITÃO, 2016). O uso de um processo de desenvolvimento de software sólido, como as metodologias Agile e DevOps, pode impedir que bugs aconteçam. O teste de qualidade está embutido nessas metodologias de desenvolvimento.

Uma dessas práticas de desenvolvimento, é o desenvolvimento orientado a testes (ROCHA, et. al., 2017). Os testes devem ser criados antes que um recurso seja codificado para fornecer um padrão contra o qual codificá-lo. Outra prática recomendada é usar o desenvolvimento orientado por comportamento, que incentiva os desenvolvedores a codificar um aplicativo e documentar o processo com base em como um usuário deve interagir com ele (LEITÃO, 2016).

Os desenvolvedores podem impedir que os bugs cheguem aos usuários testando com antecedência e frequência. Juntamente com o teste de software, uma revisão de código por pares com outros desenvolvedores, um desenvolvedor sênior ou uma equipe de garantia de qualidade (QA) pode ser útil. (ROCHA, et. al., 2017), cita que o *benchmarking* ou o teste de *benchmark* estabelecem as expectativas de desempenho de linha de base para o software em diferentes tipos de cargas de trabalho. Os testes de *benchmark* podem avaliar a estabilidade, a capacidade de resposta, a velocidade e a eficácia do software.

Bugs que podem ficar dormentes sob um conjunto de condições podem causar um problema sério em outros (DO AMARAL; SIRQUEIRA, 2022). O teste de *benchmark* pode ajudar a identificar esses bugs. Alguns tipos de *benchmarking* são os seguintes: O *benchmarking* de carga avalia os sistemas de software sob uma carga específica, que geralmente é a quantidade usual de tráfego esperada para um aplicativo (COMIS, et. al., 2022).

O *benchmarking* Spike avalia o desempenho do software durante um aumento súbito na carga de trabalho. O benchmarking de ponto de interrupção empurra um software para ver quanto estresse ele pode lidar antes de travar (LEITÃO, 2016).

Rocha et. al., (2017), cita que se um bug for encontrado no software, ele deverá ser depurado. A depuração envolve as três etapas a seguir:

* isolando o bug;
* determinando a causa raiz;
* corrigindo o problema.

A depuração é um processo de programação de computador para encontrar e resolver erros em software ou em um site, muitas vezes referidos como bugs. Muitas vezes, requer um procedimento abrangente para identificar a razão pela qual um bug ocorreu e desenvolver estratégias para garantir que um programa possa ser executado sem problemas para os usuários no futuro (COMIS, et. al., 2022).

Desenvolvedores e engenheiros de software geralmente depuram programas usando uma ferramenta digital para exibir e editar a linguagem de codificação, que contém instruções sobre como um programa funciona. A depuração permite que eles abordem seções individuais de código para garantir que cada parte de um programa opere de maneira esperada e ideal (CENDRON; ANNES, 2014).

Rocha et. al., (2017), cita que:

**Ferramenta de depuração**: Muitas vezes chamado de depurador, este software permite localizar a área em um código onde o erro ocorre e fazer os ajustes necessários. Muitas ferramentas de software de codificação vêm com ferramentas de depuração integradas.

**Exceção**: Esse é um evento que altera o fluxo esperado do código de um programa, como tentar abrir um arquivo que não existe em um programa. Em algumas circunstâncias, o programa pode manipular a exceção e continuar a ser executado, mas se a exceção interromper o programa, você normalmente iniciará um processo de depuração.

**Interface de programação de aplicativos**: Normalmente referida como API, essa ferramenta permite que diferentes linguagens de codificação operem em um programa. Por exemplo, uma API da Internet pode permitir que um usuário faça login em um site usando uma conta de outro.

**Compilador**: Este é um programa que traduz uma linguagem de codificação em um formato utilizável, dando-lhe a capacidade de depurar o programa. Ele pode identificar alguns erros no código e oferecer soluções, mas você pode precisar de uma ferramenta de depuração para resolvê-los.

**Variável**: Uma variável refere-se a como você categoriza o armazenamento no código de um programa. Você pode organizar variáveis no código para executar ações diferentes, como usar um para um conjunto de números para incluí-lo em uma linha de código.

**Valor**: Isso representa o nome específico que você atribui a uma variável depois de criá-la. Por exemplo, o valor de um conjunto de números pode ser "informações da folha de pagamento".

**Função**: Uma função descreve como as variáveis se traduzem em ação no código de um programa. As funções permitem que um programa opere da maneira que os desenvolvedores e engenheiros de software pretendem.

**Ponto de interrupção**: As pessoas normalmente usam pontos de interrupção ao investigar uma exceção ou um evento semelhante durante o processo de depuração, pois você pode usá-los para impedir que o código seja executado em seções que contenham um erro identificado anteriormente. Isso pode economizar o tempo do programador, pois eles não precisam esperar que seções de código que não estão relacionadas ao erro sejam processadas.

**Entrada**: Isso se refere à linguagem de codificação que instrui um programa a concluir uma determinada ação ou função. Você pode inserir uma linguagem de codificação em um compilador para ver se um programa funciona.

**Saída**: Isso representa os dados codificados que um programa cria depois de executá-lo usando um compilador e normalmente parece diferente da entrada devido a como as funções no código interagem. As pessoas geralmente depuram um programa para fazer com que a saída reflita um arranjo esperado de dados.

Pode ser difícil para os programadores que escreveram um pedaço de código refazer seus passos e examinar linhas de código complexas e densas. Um programa de recompensa de bugs é uma maneira de *crowdsource* um esforço de depuração. Com o *crowdsourcing*, os pesquisadores de segurança de software e hackers éticos são recompensados por encontrar problemas e fornecer relatórios de bugs que reproduzem ou mitigam a vulnerabilidade (DO AMARAL; SIRQUEIRA, 2022).

As organizações que procuram minimizar bugs de software devem equilibrar o número de implementações e reversões de versões de software que fazem. Ao fazer isso, eles garantem que o processo de depuração não atrapalhe um cronograma de lançamento de software consistente. Isso geralmente é o que as organizações que trabalham em um ambiente de desenvolvimento ágil fazem (COMIS, et. al., 2022).

No entanto, alguns bugs chegam ao produto lançado. As equipes de desenvolvimento podem tratar uma versão como parte do processo de depuração, coletando feedback sobre ela, falhando rapidamente e fazendo melhorias (DE AGUIAR, 2016). Uma equipe ou um indivíduo da equipe pode agendar um horário fixo todos os dias para resolver bugs de software (JÚNIOR et. al., 2017). Dessa forma, a coleta de dados sobre bugs e o próprio processo de depuração passam a fazer parte da programação diária. Uma equipe pode usar dados sobre o processo de depuração para estimar quanto tempo uma determinada correção levará e organizar seus esforços de acordo (ROCHA, et. al., 2017).

É impossível corrigir todos os bugs de uma só vez, e leva tempo para coletar os dados necessários para criar estimativas precisas de bugs (SOUZA, 2018). Os programadores diferem em níveis de habilidade e capacidades. E as estimativas de correção de bugs também podem variar entre os programadores que trabalham em diferentes países. Com o tempo, uma equipe pode desenvolver estimativas de benchmark para quantos bugs pode corrigir em um mês (BARICHELLO, 2018).

A depuração nunca é perfeita ou completa (GOMES et. al., 2008). Novos bugs sempre aparecem. As equipes de desenvolvimento devem ter como objetivo abordar os bugs de forma eficiente e fornecer valor líquido positivo para as partes interessadas com cada versão de software (DE AGUIAR, 2016).

**REFERÊNCIAS BIBLIOGRÁFICAS**

ANDRADE, Carlos AR; SANTOS, André LM; BORBA, Paulo HM. AspectH: **Uma Extensão Orientada a Aspectos de Haskel**. Revista de Informática Teórica e Aplicada, v. 11, n. 2, p. 21-32, 2004. Disponível em: <https://www.seer.ufrgs.br/rita/article/view/rita_v11_n2_p21-32> Acesso em: 08 de março de 2023.

BARICHELLO, Leonardo. **Programação de computadores em scratch por meio de jogos**. Computação na Educação Básica: Fundamentos e Experiências, 2018. Disponível em: <https://books.google.com.bo/books?hl=pt-BR&lr=&id=EBvLDwAAQBAJ&oi=fnd&pg=PT280&dq=programa%C3%A7%C3%A3o+evitando+bugs&ots=3Z4rzTSHN3&sig=6XeQA1FxnU1M-v5Srpuu23vAsos> Acesso em: 04 de abril de 2023.

BARROS, Gabryela Santana et al. **Análise experimental entre as técnicas TDD e Test-Last no processo de manutenção corretiva de software**. 2019. Disponível em: <http://ri.ucsal.br:8080/jspui/handle/prefix/879> Acesso em: 08 de março de 2023.

BELL, Jonathan et al. DeFlaker: **Detectando automaticamente testes escamosos**. In: Anais da 40ª Conferência Internacional de Engenharia de Software. 2018. p. 433-444. Disponível em: <https://dl.acm.org/doi/abs/10.1145/3180155.3180164> Acesso em: 04 de abril de 2023.

BELLER, M., GOUSIOS, G., PANICHELLA, A., ZAIDMAN, A., 2015. **Quando, como e por que os desenvolvedores (não) testam em seus ides**. In: Anais do 10º Encontro Conjunto de Fundamentos da Engenharia de Software (ESEC/FSE) 2015. ACM, pp. 179–190. Disponível em: <https://dl.acm.org/doi/10.1145/2786805.2786843> Acesso em: 04 de abril de 2023.

BELLER, M., SPRUIT, N., SPINELLIS, D., ZAIDMAN, A., 2018. **Sobre a dicotomia do comportamento de depuração entre programadores**. In: Anais da 40ª Conferência Internacional de Engenharia de Software (ICSE). ACM, pp. 572–583.

BREU, S., PREMRAJ, R., SILLITO, J., ZIMMERMANN, T., 2010. **Necessidades de informação em relatórios de bugs: melhorando a cooperação entre desenvolvedores e usuários**. Em: Anais da Conferência ACM sobre Trabalho Cooperativo Apoiado por Computador (CSCW). ACM, pp. 301–310. Disponível em: <https://dl.acm.org/doi/10.1145/1718918.1718973> Acesso em: 03 de abril de 2023.

CARVALHAES, Millys Fabrielle Araujo; VIEIRA, Gonçalves Matheus; ESTALBAR, Junior Augusto Xavier. **Desenvolvimento de um sistema de software para gestão de serviços eprodutos de um pet shop**. 2021. Disponível em: <http://repositorio.aee.edu.br/bitstream/aee/19679/1/Entrega%2005%20%284%29.pdf> Acesso em: 08 de março de 2023.

CATOLINO, Gemma et al. ***Not all bugs are the same: Understanding, characterizing, and classifying bug types***. Journal of Systems and Software, v. 152, p. 165-181, 2019. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0164121219300536> Acesso em: 04 de abril de 2023.

CENDRON, Paulo Roberto; ANNES, Ricardo. **Programação de computadores para jovens**. Extensão Tecnológica: Revista de Extensão do Instituto Federal Catarinense, n. 2, p. 105-108, 2014. Disponível em: <http://publicacoes.ifc.edu.br/index.php/RevExt/article/view/84> Acesso em: 04 de abril de 2023.

CLEMENTE, Bianca Cristina et al. **Caça objetos: reconhecimento de imagem como um jogo**. 2021. Disponível em: <http://ric.cps.sp.gov.br/handle/123456789/6712> Acesso em: 08 de março de 2023.

COMIS, Diego et al. **Avaliação das ferramentas de análise estática de códigos PHP utilizando templates de websites eGov baseados em CMS**. In: Anais da VI Escola Regional de Engenharia de Software. SBC, 2022. p. 171-180. Disponível em: <https://sol.sbc.org.br/index.php/eres/article/view/22384> Acesso em: 04 de abril de 2023.

COSTA, José. **Introdução à Programação Funcional com Haskell**. Porto Alegre: Bookman, 2013.

DE AGUIAR, Rafael Azevedo. **Um estudo sobre a incidência de bugs relacionados a deadlocks em aplicações C# de código aberto**. 2016. Tese de Doutorado. Universidade Federal de Pernambuco. Disponível em: <https://www.cin.ufpe.br/~tg/2016-1/raa7.pdf> Acesso em: 04 de abril de 2023.

DE MATOS, Altieres et al. **Estudo preliminar sobre os impactos do desenvolvimento orientado a testes no processo de revisao de código**. In: Anais da II Escola Regional de Engenharia de Software. SBC, 2018. p. 147-156. Disponível em: <https://sol.sbc.org.br/index.php/eres/article/view/10068> Acesso em: 08 de março de 2023.

DE MATOS, Altieres et al. **Estudo preliminar sobre os impactos do desenvolvimento orientado a testes no processo de revisao de código**. In: Anais da II Escola Regional de Engenharia de Software. SBC, 2018. p. 147-156. Disponível em: <https://sol.sbc.org.br/index.php/eres/article/view/10068> Acesso em: 08 de março de 2023.

DE SOUZA, JHONATA LIMA. **A Contribuição do Low Code no Âmbito Educacional: Um Mapeamento Sistemático da Literatura**. Disponível em: <https://www.cin.ufpe.br/~tg/2021-2/tg_SI/TG_jls3.pdf> Acesso em: 08 de março de 2023.

DEJEU, Jan. **Haskell School of Music: De Sinais a Sinfonias**. Yale University Press, 2013.

DO AMARAL, Luiz Fernando Benini; SIRQUEIRA, Tassio Ferenzini Martins. **Uma Análise das Métricas da Evolução do Código Fonte do Joomla**. Caderno de Estudos em Engenharia de Software, v. 3, n. 2, 2022. Disponível em: <http://seer.uniacademia.edu.br/index.php/engsoftware/article/view/3014> Acesso em: 04 de abril de 2023.

FONSECA, Rodrigo; SANTOS, Lucas. **Programação Funcional com Haskell**. Florianópolis: Editora da UFSC, 2016.

GOMES, Anabela; HENRIQUES, Joana; MENDES, António José. **Uma proposta para ajudar alunos com dificuldades na aprendizagem inicial de programação de computadores**. Educação, Formação e Tecnologias, v. 1, n. 01, p. 93-103, 2008. Disponível em: <http://educa.fcc.org.br/scielo.php?pid=S1646-933x2008000100009&script=sci_abstract&tlng=pt> Acesso em: 04 de abril de 2023.

HAZIN, Raphael et al. **Técnicas e Padrões para Otimizar a Interpretação de Códigos Fontes Aplicados à Fábrica de Software**. 2017 - Disponível em: <https://www.researchgate.net/profile/Washington-Almeida-2/publication/334684379_Tecnicas_e_Padroes_para_Otimizar_a_Interpretacao_de_Codigos_Fontes_Aplicados_a_Fabrica_de_Software/links/5d3a1caea6fdcc370a60492f/Tecnicas-e-Padroes-para-Otimizar-a-Interpretacao-de-Codigos-Fontes-Aplicados-a-Fabrica-de-Software.pdf> Acesso em: 08 de março de 2023.

JÚNIOR, Piancó et al. **Analisando as relações entre mudanças no código fonte e bugs no software**. 2017. Disponível em: <http://www.repositorio.ufal.br/handle/riufal/2922> Acesso em: 04 de abril de 2023.

LEITÃO, Wellyngton Amaral. **Estudo exploratório de bugs de tratamento de exceção na plataforma android: estudo exploratório**. 2016. Disponível em: <https://repositorio.ufc.br/handle/riufc/24779> Acesso em: 04 de abril de 2023.

LUNA, Ryan Camargo. **Agrupamento de bugs de software a partir do modelo de linguagem pré-treinado BERT e métodos de clustering**. 2022. Trabalho de Conclusão de Curso. Universidade Tecnológica Federal do Paraná. Disponível em: <http://repositorio.utfpr.edu.br/jspui/handle/1/29092> Acesso em: 04 de abril de 2023.

MAGUIRE, Sandy. **Pensando com tipos.** “*Type-Level Programming in Haskell*”, 1º Ed. 2018.

MANGRICH, Eduardo de Souza et al. **Projeto e implementação de chatbot no processo de autorização de parcelas para pagamentos na indústria da construção**. 2021. Disponível em: <https://repositorio.ufsc.br/handle/123456789/230478> Acesso em: 08 de março de 2023.

MARTIN, Robert C. **Código limpo: habilidades práticas do Agile software**. Alta Books, 2019.

MEIRELES, Daniel Carlos; DE RESENDE COSTA, Romualdo Monteiro. **Desenvolvimento de uma Aplicação Baseada em Mobile Crowd Sensing para a Aquisição de Bens de Consumo Não Duráveis**. Caderno de Estudos em Sistemas de Informação, v. 7, n. 2, 2022. Disponível em: <http://seer.uniacademia.edu.br/index.php/cesi/article/view/2655> Acesso em: 08 de março de 2023.

MENEZES, Ana Paula. **Haskell: Uma Abordagem Prática**. São Paulo: Novatec, 2012.

PIERCE, Benjamin C. **Types and programming languages**. MIT press, 2002. Disponível em: <https://books.google.com.bo/books?hl=pt-BR&lr=&id=ti6zoAC9Ph8C&oi=fnd&pg=PR13&dq=Types+and+Programming+Languages%22+de+Benjamin+C.+Pierce&ots=EDM9yEi1VF&sig=lu5TzfhCzo_8POU7x0iWB10sA1M> Acesso em: 3 de abril de 2023.

PIERCE, Benjamin C. **Types and Programming Languages: The Next Generation**. In: LICS. 2003. p. 32-. Disponível em: <https://www.academia.edu/download/42906352/tng-lics2003-slides.pdf> Acesso em: 3 de abril de 2023.

ROCHA, Kátia Coelho da; BASSO, Marcus Vinicius de Azevedo. **Teoria dos Campos Conceituais na análise de programação em Scratch**. RENOTE: revista novas tecnologias na educação. Porto Alegre. Vol. 15, n. 2 (dez. 2017), 10 p., 2017. Disponível em: <https://www.lume.ufrgs.br/handle/10183/223836> Acesso em: 04 de abril de 2023.

ROCHA, Ricardo L.; NETO, João José. **Uma proposta de linguagem de programação funcional com características adaptativas**. In: IX Congreso Argentino de Ciencias de la Computación. 2003. Disponível em: <http://sedici.unlp.edu.ar/bitstream/handle/10915/22812/Documento_completo.pdf?sequence=1> Acesso em: 08 de março de 2023.

SARCINELLI, Bruno Cezar Scopel. **Avaliação da ferramenta Minitest no desenvolvimento guiado por testes do framework Ruby on Rails**. O Comunicante, v. 8, n. 2, p. 19-30, 2018. Disponível em: <http://ebrevistas.eb.mil.br/OC/article/view/1745> Acesso em: 08 de março de 2023.

SCOTT, Michael Lee. ***Programming language pragmatics***. Morgan Kaufmann, 2000. Disponível em: <https://books.google.com.bo/books?hl=pt-BR&lr=&id=To3xpkvkPvMC&oi=fnd&pg=PR17&dq=Programming+Language+Pragmatics%22+de+Michael+L.+Scott&ots=9eCNospARR&sig=z7PZMkEEoZ2C-BJPSzSAXrFY4Uc> Acesso em: 3 de abril de 2023.

SILVA, Allysson Costa et al. **Vetores de Parágrafo aplicados à localização de características e bugs de software**. 2020. Disponível em: <http://repositorio.ufu.br/handle/123456789/29339> Acesso em: 04 de abril de 2023.

SILVA, Antunes Dantas da et al. **Um estudo sobre diferentes linguagens de programação para introdução da programação funcional**. 2019. Disponível em: <http://dspace.sti.ufcg.edu.br:8080/jspui/handle/riufcg/20352> Acesso em: 08 de março de 2023.

SOUZA, William Amaral de. **Uma análise do uso da plataforma MIT APP INVENTOR 2 como ferramenta para auxiliar no processo de ensino e aprendizagem de programação**. 2018. Disponível em: Acesso em: 04 de abril de 2023.

TAKAGI, Jennifer Camila dos Reis. **Especificação de parâmetros para o uso da ferramenta Eslint em projetos React JS para manutenção de código limpo**. 2021. Disponível em: <https://repositorio.ifsc.edu.br/bitstream/handle/123456789/2031/TCC_Jennifer-Takagi.pdf?sequence=1> Acesso em: 08 de março de 2023.

THOMPSON, Simon. ***Type theory and functional programming***. Addison Wesley, 1991. Disponível em: <https://kar.kent.ac.uk/20998/1/ttfp.pdf> Acesso em: 3 de abril de 2023.

VEDOVATTO, Thiago; COSTA, Esdras Teixeira. **Haskell: uma linguagem de programação ideal para matemáticos**. 2013 - Disponível em: <https://www.researchgate.net/profile/Thiago-Vedovatto/publication/267810885_Haskell_uma_linguagem_de_programacao_ideal_para_matematicos/links/545ad7d40cf25c508c319dbc/Haskell-uma-linguagem-de-programacao-ideal-para-matematicos.pdf> Acesso em: 08 de março de 2023.

WAGNER, Daniel Pinheiro; FANTONI, Felipe Silva. **Aplicando Desenvolvimento Guiado por Testes para Auxiliar na Evoluç ao de Arquiteturas Monolıticas para Microsserviços**. – 2021 - Disponível em: <http://bib.pucminas.br:8080/pergamumweb/vinculos/000099/0000996d.pdf> Acesso em: 08 de março de 2023.

WIRED. **Os piores bugs de software da história**. Disponível em: <https://www.wired.com/2005/11/historys-worst-software-bugs/> Acesso em: 04 de abril de 2023,