WBA0883_V1.0

QUALIDADE DE SOFTWARE COM CLEAN CODE E TÉCNICAS DE USABILIDADE

Stella Marys Dornelas Lamounier

QUALIDADE DE SOFTWARE COM CLEAN CODE E TÉCNICAS DE USABILIDADE

1ª edição

São Paulo
Platos Soluções Educacionais S.A
2021

© 2021 por Platos Soluções Educacionais S.A.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Platos Soluções Educacionais S.A.

Diretor Presidente Platos Soluções Educacionais S.A
Paulo de Tarso Pires de Moraes

Conselho Acadêmico
Carlos Roberto Pagani Junior
Camila Turchetti Bacan Gabiatti
Camila Braga de Oliveira Higa
Giani Vendramel de Oliveira
Gislaine Denisale Ferreira
Henrique Salustiano Silva
Mariana Gerardi Mello
Nirse Ruscheinsky Breternitz
Priscila Pereira Silva
Tayra Carolina Nascimento Aleixo

Coordenador
Henrique Salustiano Silva

Revisor
Marco Ikuro Hisatomi

Editorial
Alessandra Cristina Fahl
Beatriz Meloni Montefusco
Carolina Yaly
Mariana de Campos Barroso
Paola Andressa Machado Leal

Dados Internacionais de Catalogação na Publicação (CIP)

Lamounier, Stella Marys Dornelas
L236q Qualidade de software com Clean Code e técnicas de usabilidade / Stella Marys Dornelas Lamounier, – São Paulo: Platos Soluções Educacionais S.A., 2021.
44 p.

ISBN 978-65-89965-56-5

1. Qualidade de software. 2. Clean code. 3. Técnicas de usabilidade. I. Título.

CDD 005

Evelyn Moraes – CRB 010289/O

2021
Platos Soluções Educacionais S.A
Alameda Santos, n° 960 – Cerqueira César
CEP: 01418-002— São Paulo — SP
Homepage: https://www.platosedu.com.br/

QUALIDADE DE SOFTWARE COM CLEAN CODE E TÉCNICAS DE USABILIDADE

SUMÁRIO

Qualidade de software: fundamentos e conformidade de requisitos

Autoria: Stella Marys Dornelas Lamounier
Leitura crítica: Marco Ikuro Hisatomi

Objetivos

- Orientar sobre aspectos relacionados à qualidade de software.
- Demonstrar os padrões que norteiam os conceitos de qualidade de software.
- Compreender requisitos e as principais diferenças de requisitos funcionais e não funcionais.
- Conhecer técnicas de levantamento de requisitos e sua documentação.

1. Introdução a qualidade de software

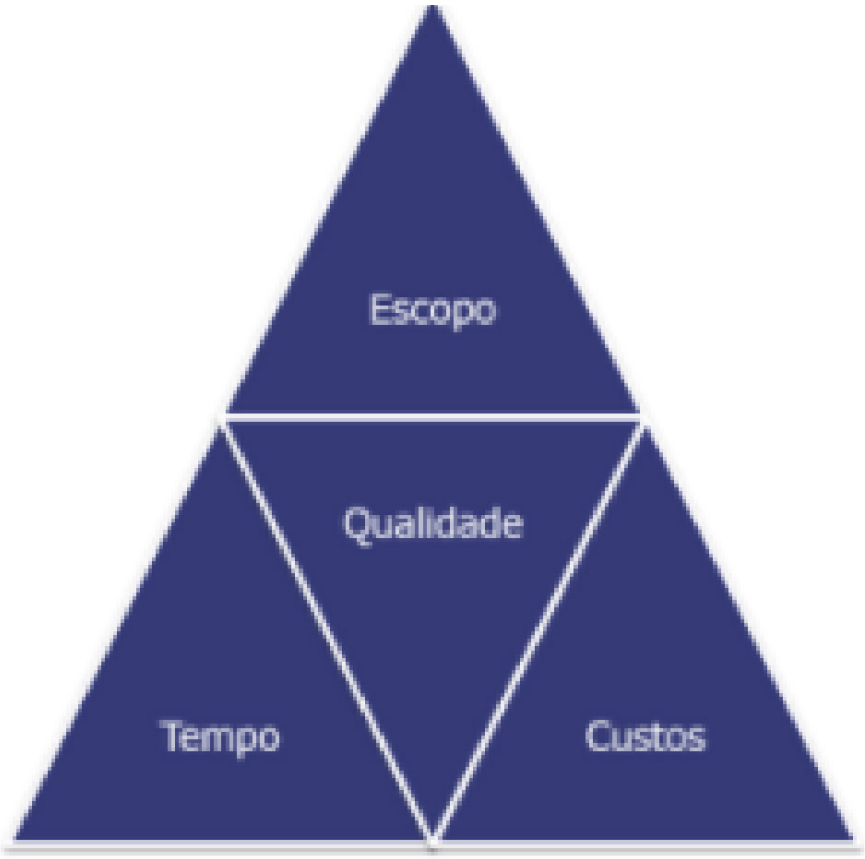
Embora a qualidade tenha ganhado campo com a implementação de padrões como a ISO – *International Organization for Standardization* (Organização Internacional de Normalização), é importante ressaltar que este conceito é muito antigo. Os egípcios já utilizavam esse conceito para estabelecer medidas para a construção de pirâmides, em que o cumprimento era dado pela medida do antebraço do faraó reinante da época. Quando havia a troca do faraó, o tamanho destas medidas também era alterado, ou seja, os operários deveriam seguir a nova medida do braço do próximo soberano. Deste modo, eram empregados bastões cortados com o cumprimento certo, tendo como unidade de medida o cúbico. Além disso, em toda lua cheia era feita uma comparação com o bastão utilizado com o padrão real, a fim de garantir a qualidade na medição para a construção precisa das pirâmides de ordem 0,05%.

Entretanto, existem relatos sobre qualidade na Grécia Antiga com a construção dos grandes templos, as navegações do século XVI, as gigantescas catedrais medievais. Todos esses feitos eram dotados de técnicas e ferramentas simples, sem a presença de instrumentos ou métodos sofisticados.

Segundo Tolezano (2018), o termo qualidade é tratado como muito subjetivo e relativo, ou seja, o que é qualidade para uma pessoa pode ser a falta de qualidade para outra. Vejamos um exemplo: um carro popular é infinitamente inferior ao um veículo de luxo, no quesito sofisticação, mas os dois podem levar ao mesmo destino. O que diferencia é a percepção do usuário, uma vez que o usuário do veículo popular pode estar extremamente satisfeito com ele, semelhantemente ao proprietário do veículo de luxo. Ambos com suas particularidades, como conforto, economia, segurança etc.

Tratando-se de qualidade que envolve sistemas computacionais, pode seguir a mesma premissa acima, em que muitas vezes o desenvolvedor é capaz de criar sistemas extremamente complexos e robustos que o usuário final mal consegue realizar um controle de mercadorias ou estoque. Ou ainda, o software pode conter tanta funcionalidade que o cliente também não consegue arcar com os custos de manter um software deste tamanho. Portanto, três atributos devem ser levados em conta para uma construção com qualidade: custos; tempo; e escopo. De acordo com Montes (2020), e conforme mencionado no triângulo das restrições do PMI–*Project Management Institute* (ou Instituto de Gerenciamento de Projetos), observe a ilustração da Figura 1.

Figura1 – Restrição Tripla



Fonte: Montes (2020, [s.p.]).

A partir desse conceito, a qualidade é alcançada quando as necessidades do cliente devem estar implícitas ou explicitas verdadeiramente no escopo (escopo), ser disponibilizado no prazo esperado (tempo) com

preço e custo compatível (custo). Além disso, com o aumento do escopo, deverá haver estudos para ajustar o tempo e o custo, sendo que para diminuir prazo, automaticamente o custo deverá ser revisto, tendo em vista que reflete em questões como a mão de obra. Por fim, para a diminuição de custo, é preciso levar em consideração uma reanálise de escopo.

1.1 Crise do software

Durante década de 1950, acreditava-se que o desempenho do computador seria proporcional ao quadrado do seu preço, conhecida como lei de Grosch. Era comum na época os usuários adquirirem computadores de grande porte ou alugar máquinas diretamente do fabricante, o que levava à seguinte conclusão: “problemas maiores, máquinas maiores” (KOSCIANSKI; SOARES, 2007).

Ainda, de acordo com os autores, tais máquinas eram lentas, caras e com um alto consumo de energia, e só a partir do surgimento dos microprocessadores em silício que houve um efeito na produção de software. Mas, com um grande agravamento, programadores não detinham de ferramentas que medissem a qualidade dos sistemas construídos, e mal se falava em ciclos de vida de um software.

A figura do programador não era tratada como profissão. Sistemas complexos eram criados sem cronograma ou planejamento, orçamentos extrapolados era normal. Com isso, eram construídos sistemas repletos de dificuldades que não operavam corretamente, e, ainda, por muitas vezes, paravam de funcionar. Deste modo, seus desenvolvedores acabavam abandonando o projeto muito antes da sua entrega final.

Alguns sistemas da época causaram enormes problemas que custaram a vida das pessoas, como o ocorrido com o equipamento de radioterapia Therac-25. Devido a um erro de cálculo no software, seis pacientes

morreram por overdose de radiação, somado à falta de testes e documentação insuficiente sobre seu código e entendimento de erros.

Outro exemplo de destaque na época foi o famoso “Bug do milênio”, que trouxe um prejuízo de 500 bilhões de dólares para as empresas. Sua causa se deu pelo fato de a economia de espaço de armazenamento dos computadores e softwares ligados muitas vezes armazenavam anos para datas com números de dois dígitos, como 99 para 1999. Esses softwares também interpretavam 00 para significar 1900, em vez de 2000; por isso, quando chegasse ano de 2000, os bugs apareceriam.

Neste cenário é sabido que o software já tem seu papel importante na sociedade, haja vista que sua criação repleta de erros/defeitos pode agravar situações que envolvem até vidas humanas, como o ocorrido com o software Patriot, em 1991, quando um erro matemático de arredondamento numérico calculou, de formar incorreta, o tempo de projeção para detectar e atacar mísseis. Deste modo, o Patriot ignorou a presença de mísseis inimigos acarretando a morte de 28 soldados e mais de 100 feridos.

Um outro exemplo, somando um prejuízo financeiro, foi uma falha no recurso de “visualizar como”, do Facebook, em 2018, permitindo o roubo das chaves digitais na rede social. Os dados de mais de 50 milhões de usuários ficaram disponíveis, possibilitando, assim, a invasão de contas e exposição de dados pessoais. Com isto, este ataque ocasionou para a empresa um prejuízo de 13 bilhões de dólares em valor de mercado.

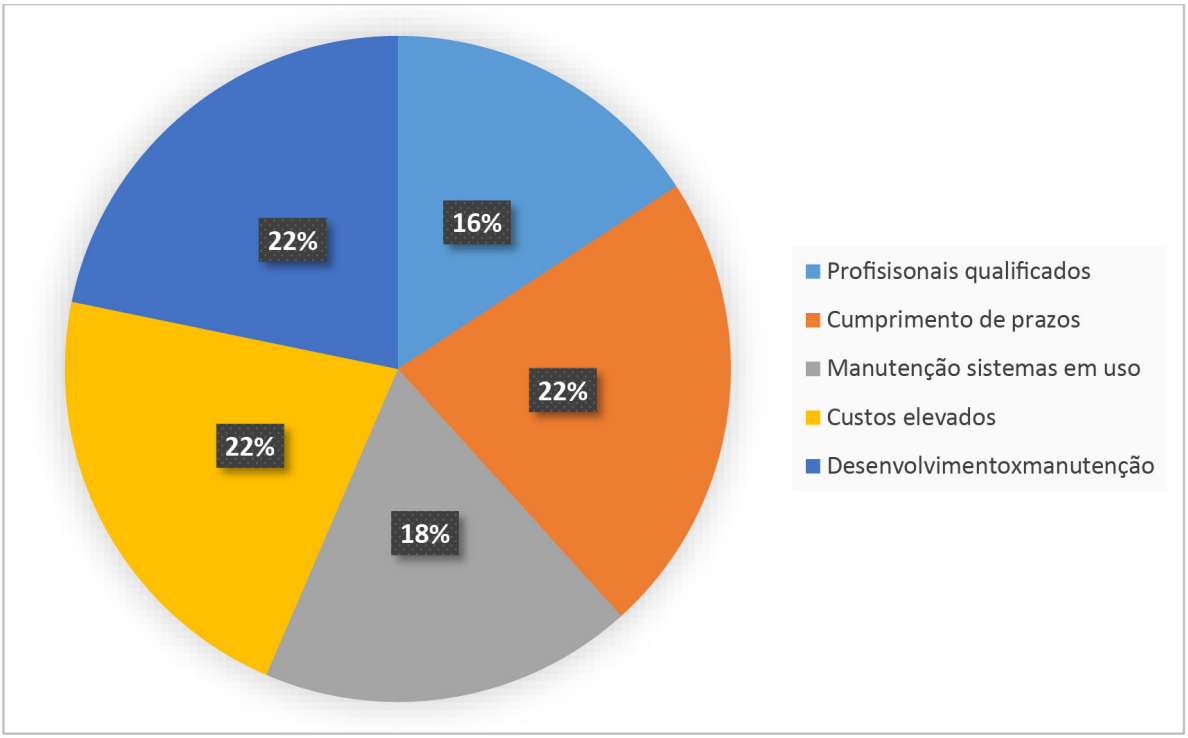
Neste contexto, por volta de 1970, culminou a chamada “Crise do Software”, quando o número de sistemas cresceu exponencialmente, mas, ao mesmo tempo, sua manutenção era quase impossível. Muitos profissionais da área denominavam essa crise como:

> O temo “Crise do Software”, começou a ser utilizados por volta da década de 60, tem historicamente aludido a um conjunto de problemas recorrentes enfrentados no desenvolvimento do software (construção,

> implantação e manutenção) de software. Abrange todos os problemas relacionados a: como sistemas computacionais são construídos, como sistemas computacionais são implantados, (REZENDE, 2006, p. 8)

Vale ressaltar que a Crise do Software não está apenas relacionada a sistemas que não funcionam, mas também à substituição de sistemas antigos e obsoletos por software mais modernos, com o mínimo de erros possíveis, sem que haja prejuízos para a empresa. O Gráfico 1 ilustra os principais problemas do desenvolvimento de software.

Gráfico 1 – Problemas com desenvolvimento de software



Fonte: adaptado de Cabral et al. (2014, p. 3).

E só a partir desse marco histórico é que profissionais da área de tecnologia perceberam o quão era importante aprimorar e documentar métodos e padrões para diminuir custo e tempo, criando sistemas mais produtivos e com maior qualidade. Tudo isso fez surgir a Engenharia de Software, responsável por aplicações sistemáticas e disciplinadas na busca qualitativa para desenvolvimento, operação e manutenção do software.

A Engenharia de Software está fortemente ligada à qualidade de sistemas, pois trata-se de metodologia com soluções para a construção de sistemas que obedece a processos e padrões de qualidade, desde a percepção do problema, a entrega final, e até mesmo quando ele sai de operação. Seu foco é resolver problemas a partir de roteiros e técnicas que visam a criação de software com qualidade.

1.2 Erro versus Defeito versus Falha

Independentemente da área de conhecimento, processos podem ser tratados como inofensivos ou que levem a grandes perdas. É importante ressaltar que estamos lidando com programas de computador criados por pessoas e, pessoas estão sempre propensas a falhas. Não existe um sistema com “defeito zero”; pode existir um sistema com o mínimo de erros possível que foi submetido a inúmeros testes, com várias revisões ao longo do seu processo de construção.

Porém, qual a melhor maneira de explicar um problema em um software, ou quando, de repente, o sistema para de funcionar? É preciso, primeiramente, saber distinguir erro, defeito e falha. A seguir, observe a diferença de cada um destes problemas conforme descrito por Pressman (2011).

O erro pode ser caracterizado como a causa de um defeito ou de uma falha, e é produzido a partir de uma ação humana que produz um resultado incorreto. Ele está presente na manipulação de dados, no código fonte, na própria documentação, ou em testes quando não executados de acordo com sua política. Podemos citar como exemplo um retorno de um valor como null dado como erro que, posteriormente, ocasionou uma falha de software.

Um sistema apresenta algum defeito quando é diagnosticado sua imperfeição ou má implementação no código. Entretanto, um defeito não está apenas ligado ao que faz um programa parar de funcionar, mas

também ao “que não funciona conforme deve ser”. Um defeito pode ser ocasionado por vários motivos, como: má especificação de requisitos; inconsistência de dados; ou até uma má interpretação de requisitos por parte dos analistas.

Um defeito muito comum é ocasionado por divisões por zeros, havendo um estouro de campo no algoritmo, mal tratamento de dados (datas inválidas), documentação incompleta/desatualizada. Lembre-se que um sistema pode estar repleto de defeitos que por muitas vezes estão invisíveis aos olhos dos usuários.

Já as falhas estão relacionadas ao resultado incorreto obtido, é provocado por uma condição inesperada ou defeito. Outra consideração é que falhas podem estar ligadas a fatores externos e, não somente a linhas de programação. Por exemplo, problemas com base de dados, sistemas corrompidos ou até mesmo a uma queda repentina de energia, que dependendo do tipo de falha, o usuário pode conviver com ela perfeitamente sem que seu trabalho seja prejudicado. Ou, ainda, falhas que podem levar a catástrofes financeiras ou que coloquem em risco a vida de pessoas. A Figura 2 ilustra a familiaridade entre erro, defeito e falha.

Figura 2 – Erro versus Defeito versus Falha

Fonte: elaborada pela autora.

A figura apresenta que erros, defeito e falha se completam. Primeiramente, o erro está relacionado a uma ação humana que, posteriormente, pode produzir algum defeito no “software”, e quando o código fonte do sistema é executado ele irá apresentar uma falha.

1.3 Padrões de qualidade

Apesar da qualidade ser tratada como algo subjetivo, a todo momento buscam-se métodos que garantam a qualidade em produtos ou software, formas e métodos capazes de estabelecer conceitos, métricas e padrões para a criação de produtos ou softwares que gerem sempre a satisfação por parte do cliente.

Ao longo dos anos, foram criadas normas internacionais para garantir e padronizar serviços ou produtos em diferentes países. Uma delas é a norma internacional ISO, nascida em 1946 em uma conferência em Londres (Reino Unido). De modo geral, a ISO tem como funcionalidade recomendar o uso de normas internacionais padronizadas, oferecendo padrões comuns e definindo critérios de aceitações. São os chamados padrões ISO que hoje estão presentes em mais de 50 países e atingem diferentes áreas de atuação, como tecnologia, meio ambiente, alimentos, saúde etc.

Sua utilização em países distintos é de suma importância para os usuários, pois é a partir destas normas que eles podem encontrar compatibilidades em aparelhos de tecnologias, produtos, como normas para montadoras de veículos, segurança do trabalho, boas práticas de higiene etc.

Não diferente a isso, softwares também estão relacionados a normas internacionais de padrões ISO, mais ligadas a documentos informativos do que a regulamentações, o que por muitas vezes não se pode aplicar alguma sansão caso venha sofrer um grande impacto como acontece nas Engenharias. O Quadro 1 ressalta as principais normas ISO de software, e note que a maioria está fortemente ligada à qualidade do produto de software, ou seja, há uma grande preocupação em produzir sistemas com qualidade e que atendam as especificações estabelecidas.

Quadro 1 – Normas ISO para software

Norma	Propósito
ISO 12207	Processo de ciclo de vida de software.
ISO/IEC 12119:1994	Pacotes de software – Requisitos de qualidade e testes.
ISO/IEC 14598-1:1999	Avaliação de qualidade de produto de software.
ISO/IEC 9126-1: 2001	Modelo de qualidade – Características.
ISO/IEC 25000:2005	Modelo de qualidade de software, nova versão da série 14.598 e 9.126.
ISO/IEC 9241: 1998	Ergonomia de software.
ISO/IEC 20926:2003	Medida de software por pontos de função.
ISO/IEC 90000:-3:2004	Diretivas para a aplicação da ISO 9001 ao software.
ISO/IEC 9001:20000	Requisitos para sistemas de gerenciamento de qualidade para empresas de software ou não.

Fonte: Koscianski e Soares (2007, p. 54).

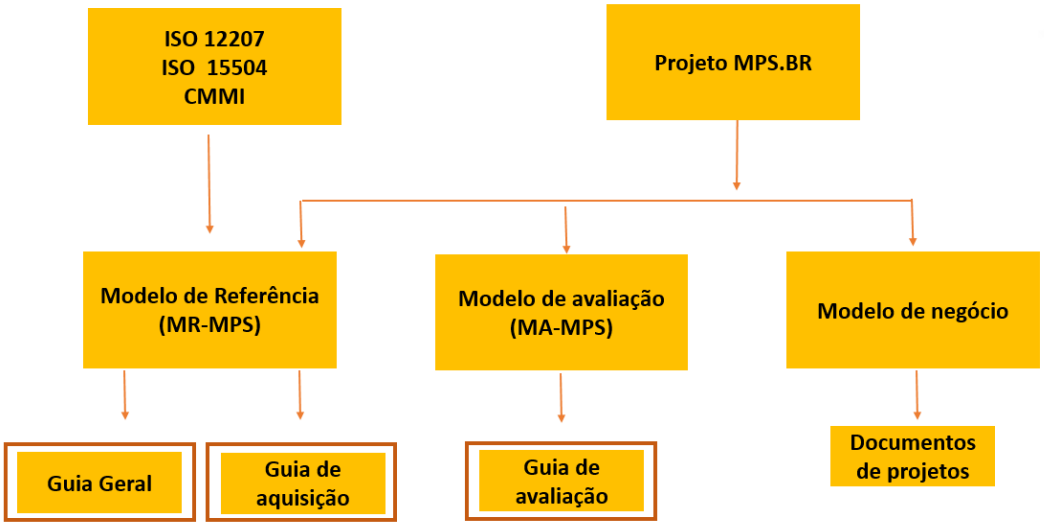
Existem, ainda, organizações nacionais que tratam padrões de qualidade de software, como a Associação Brasileira de Normas Técnicas (ABNT), norma que rege e representa no Brasil a ISO e a IEC (Comissão Eletrotécnica Internacional ou *International Electrotechnical Commission*), que também abrange diversas áreas como construção civil, tecnologias, agricultura, engenharias, entre outros.

Temos também a MPS.BR (Melhoria do Processo de Software Brasileiro), criada em 2003 pelos órgãos SOFTEX, COPPE/UFRJ, CESAR, CenPRA e CELEPAR. Seu foco principal micro são as pequenas e médias empresas de software brasileiras que detêm pouco recurso para melhoria de software brasileiro, uma vez que as certificações ISO muitas vezes são inviáveis pelo seu alto valor de aquisição. Tal fato pode ser um

impedimento a essas empresas a fim de concorrer no mesmo nível com as grandes multinacionais de desenvolvimento de sistemas.

A MPS.BR é usada por empresas brasileiras e algumas que fazem parte do Mercosul. Tal norma segue os mesmos padrões e abordagens dos modelos internacionais para a melhoria nos processos de software, mas com um percentual de 40 inferior na hora de adquirir uma certificação. A Figura 3 ilustra como é a estrutura hierárquica da MPS.BR e as regidas por elas.

Figura 3 – Hierarquia MPS.BR



Fonte: adaptada de Weber et al. (2006, p. 3).

Notem que a MPS.BR advém dos principais órgãos internacionais ISO e CMMI, e ambos tratam sobre a melhoria no processo de qualidade de um produto ou software. O Modelo Integrado de Maturidade em Capacitação (CMMI) é uma norma criada pela SEI, servindo como um manual, a fim de guiar seus usuários na melhoria contínua de processos em projetos, e auxiliar com práticas que foram implementadas de forma bem-sucedida, fornecendo, assim, maturidade no desenvolvimento ou manutenção de produtos/software. O CMMI é dividido em cinco níveis de maturidade: 1) inicial; 2) gerenciado; 3) definido; 4) quantitativamente gerenciado; 5) Otimização.

Quanto maior o nível da maturidade, mais perto do topo a empresa consegue chegar, indicando que passou por várias fases até sua chegada ao topo, garantindo que, em cada etapa, processos devem ser cumpridos para que os sistemas produzidos possuam sempre qualidade em seus processos.

Similar ao CMMI, o MPS.BR também é dividido por níveis de maturidade, seguindo essa mesma premissa, quanto mais alto, maior é o seu grau de maturidade, conforme apresentado na Figura 4.

Figura 4 – Níveis de maturidade MPS.BR

Fonte: adaptada de Softex e MPS.BR (2012, p. 18).

As organizações que adotam padrões de qualidade, seja por ISO, CMMI ou MPS-BR tendem a aumentar o poder de gerenciamento das atividades no controle da qualidade de seus processos e de produtos. Esses modelos de maturidade podem prever procedimentos com foco em UX e Clean Code, respectivamente, em processos de gestão de regras de negócio e de construção do código-fonte. Dessa forma, proporciona maior sucesso em critérios de qualidade do software.

2. Requisitos

Implantar normas e processos dentro de uma empresa de desenvolvimento de software, talvez não seja um trabalho fácil, mas, dependendo dos procedimentos adotados, podem se tornar atividades complexas quando não há uma estimativa de quão é importante estabelecer as técnicas apropriadas para a busca e levantamento de requisitos.

Saber definir de maneira precisa quais são os requisitos de um software tornam a criação de um sistema com mais qualidade e menos suscetibilidade a erros, que podem ocasionar falhas em funcionalidades. E até mesmo a falta de usabilidade do software pode atrapalhar todo o funcionamento se não forem detectados precocemente, ou seja, no início da criação do projeto.

Portando, sem essa definição e planejamento, a perda de tempo é inevitável, assim como o aumento de custos e problemas na construção do software. Com isso, a garantia de qualidade final do produto a ser construído é comprometida, isto é, sem as características e funcionalidades que o cliente deseja.

Inicialmente, deve-se ter um conhecimento sobre o que é um requisito de software:

> Pode ser definido como qualquer coisa que alguém queira. No contexto de desenvolvimento de software, entende-se por requisitos as propriedades que o sistema (ainda em projeto) devem manifestar quando estiverem em desenvolvidos. Os requisitos expressam as necessidades dos usuários e as restrições que são apresentadas a um sistema que devem ser consideradas durante o desenvolvimento. (REZENDE, 2006, p. 66)

Basicamente, os requisitos podem ser divididos de acordo com as necessidades do usuário. Independentemente do seu tipo, o requisito

deve ser representado de forma simples e clara, ou seja, de fácil interpretação para quem irá implementar, a fim de compreender informações demonstradas de acordo com a regras de negócio da empresa ou até mesmo a experiência do usuário. Sempre que possível, utilizar uma linguagem mais natural.

2.1 Requisitos funcionais

Este modelo de requisitos descreve as funcionalidades que o sistema deve ter quando estiver pronto para operar, e não está relacionado ao código fonte, tecnologias, banco de dados ou linguagens de programação. Ele é responsável por especificar os serviços e as funcionalidades que um software oferece, documentar entradas e saídas do sistema, seu comportamento em diferentes situações e principalmente o que o sistema não deve fazer. Para seu bom entendimento, todos os seus serviços devem estar muito bem definidos e jamais ter definições contraditórias.

Pode-se afirmar que requisitos funcionais são condições necessárias e obedecidas para chegar a um objetivo proposto, ou seja, demandam o que tem que ser fazer em um projeto de software e devem ser descritos do ponto de vista do cliente, com linguagem similar ao entendimento humano para especificar funções, como: excluir; cadastrar; buscar etc.

Geralmente, para melhor organização na documentação de requisitos é uma boa prática utilizar a sigla RF para requisitos funcionais. São exemplos de requisitos funcionais:

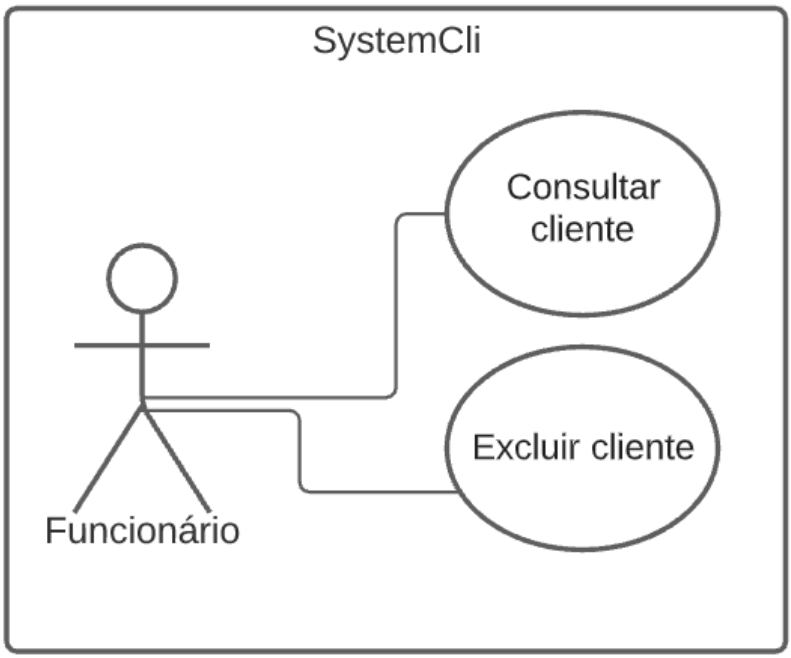
- **RF001:** o sistema deve permitir o cadastro dos fornecedores da loja.

- **RF002:** o sistema deve fornecer telas apropriadas para o usuário ler documentos disponíveis no repositório de documentos.

- **RF003:** incluir cliente como pessoa física.
- **RF004:** consultar cliente.
- **RF005:** realizar pagamento por débito e crédito.
- **RF006:** consultar estoque.
- **RF007:** excluir cliente.

Os requisitos funcionais são amplamente empregados na criação de diagramas de casos de uso, que tem como funcionalidade a criação de cenários que descrevem uma sequência de passos e funcionalidades do sistema a ser criado, facilitando o entendimento do desenvolvedor com a aplicação. A Figura 5 ilustra dois modelos de requisitos demonstrados anteriormente, o RF 004 e RF 006.

Figura 5 – Cenário SysteCLi



Fonte: elaborada pela autora.

Portanto, como dito, os requisitos funcionais não se preocupam com condições internas do sistema, ou seja, com partes que envolvem

conhecimento técnico, e sim algo que será feito, uma ação ou processo a ser realizado pelo sistema que atenda a demanda do cliente.

2.2 Requisitos não funcionais

Esses modelos de requisitos estão relacionados com as funções desempenhadas pelo sistema, suas qualidades, limitações, custo, confiabilidade, manutenibilidade, método e tempo de desenvolvimento.

Requisitos não funcionais (RNFs) estão fortemente associados à qualidade de software, e a sua não especificação pode ocasionar problemas que interferem exatamente naquilo que o cliente deseja como mencionado. Assim:

> RNFs abordam importantes aspectos relacionados à qualidade de softwares. Eles são essenciais para que softwares sejam bem sucedidos. A não observância de RNFs pode resultar em: softwares com inconsistência e de baixa qualidade; clientes e desenvolvedores insatisfeitos; tempo e custo de desenvolvimento além dos previstos devido à necessidade de se consertar softwares que não foram desenvolvidos sob a ótica da utilização de RNF. (CYSNEIROS; LEITE, 2001, p. 21)

Como boa prática de documentação de requisitos e para uma melhor organização é importante utilizar siglas, conforme mencionado nos requisitos funcionais. Aqui utilizaremos RNF. São exemplos de requisitos não funcionais:

> **RNF001:** o tempo de resposta do sistema não deve ultrapassar 30 segundos.
>
> **RNF002:** compatibilidade com sistemas operacionais Windows e Linux.
>
> **RNF003:** o sistema deve ser desenvolvido em C++.
>
> **RNF004:** o website deverá ser compatível com os navegadores Mozilla Firefox, Internet Explorer e Microsoft Edge, Chrome.

RNF005: o backup será realizado em ambiente físico e em nuvem.

Requisitos não funcionais são divididos em diversas categorias. O Quadro 2 demonstra suas principais categorias:

Quadro 2 – Modelos de RNF

Propriedade	Métrica
Velocidade/desempenho.	Transações processadas por segundo. Tempo de resposta ao usuário/evento. Tempo de atualização da tela.
Tamanho.	Kbytes, Gbytes.
Facilidade de uso.	Tempo de treinamento. Número de telas de ajuda.
Confiabilidade.	Tempo médio para falhar. Probabilidade de indisponibilidade. Taxa de ocorrência de falhas.
Interface.	Qual interface o sistema funcionará.
Hardware e software alvo.	Tipos de hardware e software que o sistema necessitará para o seu funcionamento.
Aspectos legais.	Tipos de leis ou decretos que o sistema deverá seguir.
Segurança.	Criptografia. Senhas. Certificação digital.

Fonte: elaborado pela autora.

Um dos problemas mais comuns com os requisitos não funcionais é a dificuldade em explicar, de forma exata, qual categoria pertence. Muitas vezes são dados como elementos subjetivos e vagos, que podem ser interpretados de diversas maneiras dependendo do ponto de vista analisado, o que pode gerar conflitos até mesmo com requisitos funcionais.

2.3 Documentação de requisitos

Não existe na Engenharia de Software um modelo de documento dito como certo e engessada para documentar os requisitos levantados, seja ele funcional ou não funcional. A organização deverá ser capaz de conhecer a natureza do problema a ser resolvido para sim, utilizar documentos que possam trazer mais clareza e objetividade para que os desenvolvedores entendam o que o cliente necessita. O Quadro 3 ilustra um exemplo simples e objetivo que demonstra como deve ser feito essa documentação para requisitos funcionais.

Quadro 3 – Documentação de RF

Sistema: **SISTEMA VenasMilK2000**		
Autor:	Data de criação:	Data da última atualização:
Detalhamento de requisitos.		
RF 001–Registro de vendas.		
Módulo: vendas.	Prioridade: essencial.	
Descrição: todos os produtos vendidos diariamente deverão ser registrados na memória da impressora fiscal.		
Usuário: funcionário responsável pelas vendas na empresa.		

Informações de entrada: para todos os produtos aptos para a venda, deverão ser registrados no sistema via leitura de código de barras. Cada produto deverá ter um código único, que possibilite sua leitura sem ambiguidade. Produtos que porventura necessitarem de pesagem, deverão ser pesados na presença do cliente para, então, registrados no sistema.
Informações de saída: ao final da compra é impresso um cupom fiscal contendo todos os produtos lidos pelo leitor de código de barras e suas especificações detalhadas.
Restrição: é proibido o registro de produtos sem código barras.

Fonte: elaborado pela autora.

Seguindo essa mesma linha, não existe modelo concreto a ser seguido, pois pode ser feito com os requisitos não funcionais, alterando apenas os dados que o compõem, conforme demonstrado no Quadro 4.

Quadro 4 – Documentação de RNF

Sistema: **SISTEMA VenasMilK2000**		
Autor:	Data de criação:	Data da última atualização:
Detalhamento de requisitos.		
RNF 001–Armazenamento de dados.		
Módulo: segurança.	Prioridade: essencial.	
Descrição: as vendas registradas diariamente deverão ser salvas na memória da impressora fiscal e, posteriormente, feito um backup conjunto de todos os PDV's que será armazenado em meio físico e em um servidor na nuvem.		
Responsável: funcionário responsável pelas vendas na empresa.		

Fonte: elaborado pela autora.

2.4 Técnicas de levantamento de requisitos

São encontradas inúmeras técnicas para levantar requisitos de forma clara e objetiva, a depender da dinâmica que a organização adota e seus critérios de seleção de requisitos, de acordo com o guia PMBOK (*Project Management Body of Knowledge*). Os requisitos devem ser obtidos e registrados com detalhes para uma boa execução do projeto, por meio das seguintes técnicas:

Entrevistas.
As entrevistas são técnicas muito comuns adotadas pelas empresas de desenvolvimento, com o papel de extrair informações dos clientes do projeto a ser desenvolvido, tais como: principais características; funcionalidades; saber o funcionamento da empresa; e rotina, a fim de traçar objetivos, esclarecer dúvidas e complementar requisitos.

Uma entrevista é feita de forma simples, a partir de reuniões e, que o entrevistador fará perguntas a fim de coletar informações específicas.

***Benchmarking*.**
Utilizada quando pretende-se realizar um comparativo entre métodos de trabalhos em relação às melhores práticas, a fim de identificar mudanças que elevem o grau de qualidade. Ou seja, a comparação de práticas reais com outras organizações com o objetivo de identificar as melhores práticas. Essas organizações não necessariamente devem ser externas, como a comparação de um produto pelo que vai ser lançado no mercado, ou comparação de softwares similares produzidos por empresas distintas.

Observações.
Muito utilizada para o acompanhamento das rotinas de quem vai operar o sistema, seja ele o proprietário ou funcionário da empresa. É utilizado para saber os detalhes sobre o funcionamento dos procedimentos das instituições. Ao observar a execução das tarefas, será possível perceber

detalhes que nem sempre são lembrados ou relatados pelo executor, seja ele operacional ou gestor de uma organização.

***Brainstorming*.**
Conhecido também como tempestade de ideias, as pessoas interessadas no projeto, trocam informações, falam livremente, levantando uma grande quantidade de ideias a serem desenvolvidas. As melhores são refinadas a fim de se chegar às melhores estratégias para atingir um objetivo.

Questionários e pesquisas.
Correspondem a uma técnica composta por um conjunto de perguntas criadas em um documento para a coleta de informações. Estas podem ser feitas de modo aberto ou múltipla escolha, a fim de complementar os requisitos ou quando o usuário está muito distante e não tem oportunidade para reuniões pessoais.

Análise dos documentos.
É uma técnica para levantamento de requisitos, em que a prioridade é analisar documentos da empresa. Baseia-se em fatos históricos já registrados, como também legislações vigentes, planos de negócios, contratos, outros projetos já implementados.

A partir disso, qual a melhor técnica a equipe deverá utilizar? Isso vai depender da etapa em que o projeto se encontra, da quantidade de informações disponíveis e quais as funcionalidades deseja coletar. Por exemplo, para projetos que estão sendo iniciados é mais usual as entrevistas, mapas mentais e *brainstorming*. A decisão caberá à equipe responsável pela especificação de requisitos e criação do projeto. Normalmente, adotam-se técnicas combinadas, entre duas ou mais, maximizando a possibilidade de angariar requisitos funcionais ou não funcionais mais adequados às necessidades do projeto.

Referências

CABRAL, A. A.; DA SILVA, D. B.; DE SOUZA, A. P. **A problemática do desenvolvimento de software:** crise ou calamidade crônica? Faculdades Integradas de Três Lagoas. [s.p.]. Disponível em: http://www.aems.edu.br/conexao/edicaoanterior/Sumario/2014/downloads/2014/A%20problem%C3%A1tica%20do%20desenvolvimento%20de%20softwapdf. Acesso: 15 maio 2021.

CYSNEIROS, L. M.; LEITE, J. C. S. P. **Requisitos não exigíveis:** Da elicitação ao modelo conceitual. 2001. 224 f. Tese (Doutorado em Ciências da Comutação) – Departamento de Informática, PUC-RJ, Rio de Janeiro, 2001. Disponível em: http://www-di.inf.puc-rio.br/~julio/Tese%20-%205.pdf. Acesso: 15 maio 2021.

KOSCIANSKI, A.; DOS SANTOS, M. S. **Qualidade de Software:** Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software. 2. ed. São Paulo: Novatec Editora, 2007.

MONTES, E. PMO Escritório de Projetos: Restrição Tripla. **PMO Escritório de Projetos**, [s.l.], 31 de agosto de 2020. Disponível em: https://escritoriodeprojetos.com.br/restricao-tripla. Acesso: 17 maio 2021.

PRESSMAN, R. S. **Engenharia de Software:** uma abordagem professional. 7. ed. Porto Alegre, 2011.

REZENDE, D. A. **Engenharia de software e sistemas de informação**. Rio de Janeiro: Brasport, 2006.

SOFTEX. MPSBR. **MPS – Melhoria de Processo de Software e serviços.** Guia Geral MPS de Serviços. 2012. Disponível em: http://www.softex.br/wp-content/uploads/2013/07/MPS.BR_Guia_Geral_Servicos_20121.pdf Acesso: 17 maio 2021.

TOLEZANO. J. Qualidade é um conceito bastante subjetivo, que depende muito de quem a percebe. **Qualidade,** São Paulo, 21 de maio de 2018. Disponível em: http://www.qualidade.com.br/2018/05/21/certificacao-voluntaria-de-sustentabilidade-meio-ambiente-e-um-fator-cada-vez-mais-importante-nas-decisoes-de-compra-dos-consumidores/ . Acesso em: 20 set 2021.

WEBER, K. et al. Melhoria de Processo do Software Brasileiro (MPS. BR): um programa mobilizador. *In:* **Proceedings of the XXXI Conferencia Latinoamericana de Informatica** (CLEI 2006). Santiago, agosto/2006. Disponível em: http://www.softex.br/wp-content/uploads/2015/08/Artigo_CLEI-200611.pdf. Acesso: 17 maio 2021.

Clean Code: a filosofia do “código limpo”

Autoria: Stella Marys Dornelas Lamounier
Leitura crítica: Marco Ikuro Hisatomi

Objetivos

- Demonstrar a evolução do código ao longo da história da computação.
- Demonstrar o Clean Code como filosofia na criação de código-fonte.
- Aplicar boas práticas para criação de um código limpo.
- Garantir, de forma breve, aplicações de padrões para criação de códigos com qualidade e que possam ser reutilizados ao longo do tempo.

1. Um pouco da história por trás de um código

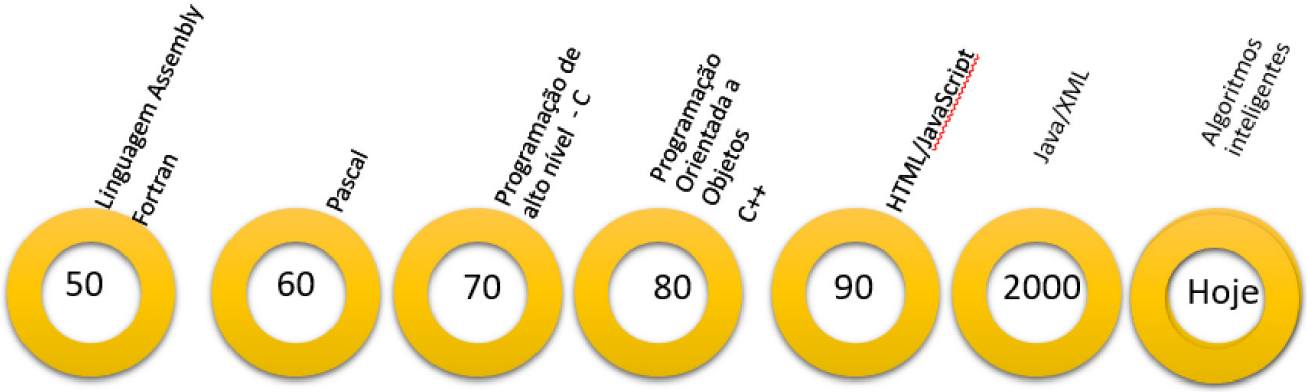
Preocupar-se com a escrita de um sistema nos dias de hoje pode parecer um pouco ultrapassado, pois já existem ferramentas que geram códigos de forma automática que não precisam mais de escrita manual. Além disso, os programadores estão muito mais preocupados com especificações e modelos de requisitos.

Engana-se quem pensa assim, pois o código-fonte representa uma linguagem rica em detalhes de cada requisito de forma bem especificada. Sempre existirá códigos, bem mais minuciosos, exatos e detalhados do que antigamente para que a máquina possa entendê-los e executá-los de forma precisa.

As linguagens de programação têm alcançado ao longo do tempo altos níveis de abstração. Com isso, surgiu a necessidade de desenvolver novos padrões de codificação que auxiliassem programadores a criar códigos de fácil entendimento e de maior qualidade. Mesmo com a presença de novas tecnologias, é impossível criar sistemas sem a percepção e a criatividade humana para um padrão ótimo de especificação de requisitos.

Muitas histórias norteiam os primeiros programas implementados, e não se sabe ao certo a data correta de sua criação, mas foi por volta da década de 1940 que se falavam em algoritmos que tratavam uma série de dados para cálculos matemáticos. Mas foi a partir da década de 1950 que as linguagens de programação começaram a evoluir consistentemente, conforme detalhado pela Figura 1 em uma linha do tempo.

Figura 1 – Linha do tempo (linguagens de programação)



Fonte: elaborada pela autora.

Ao longo de tempo, a programação também se desenvolveu, iniciando por linguagem de baixo nível, isto é, daquelas que estão mais próximas à linguagem de máquina para as linguagens de alto nível com características mais próximas a da linguagem humana, de mais fácil interpretação, com mais segurança e confiabilidade. Essa evolução também aconteceu com as linhas de código; hoje, temos códigos muito mais limpos, organizados e claros do que na década de 1950.

Portanto, mais do que se preocupar com a linguagem a ser utilizada para desenvolver um sistema computacional é saber e detalhar os requisitos, as inúmeras linhas que compõem um código-fonte, e como irão se comportar, pois, um código ruim pode até funcionar, mas dificilmente será entendível e fácil de alterar caso haja necessidade por parte da equipe ou até mesmo por parte do cliente.

1.1 O código

Construir um código-fonte requer muita competência e conhecimento técnico. E caso as técnicas confiáveis não forem seguidas, muitos problemas poderão surgir, como cronogramas extrapolados, módulos inoperantes, programas que não estão de acordo com o especificado

no levantamento de requisitos e que param de funcionar, acarretando o abandono do projeto.

Quando um programador se depara com um código “ruim” criado por outro profissional ou que ele mesmo tenha criado, e que possivelmente esqueceu como que o fez, ele possivelmente estará retardando ainda mais o processo de entrega. Quanto mais alterações o código sofrer, provavelmente mais confuso e caótico ele se tornará.

Um código, por sua vez, deve ser limpo, livre de rodeios, simples e direto, a fim possuir uma leitura facilmente compreensível por diferentes programadores. Por isso é imprescindível, também, a realização de testes de qualidade para a minimização de erros e defeitos.

De acordo com Lewis (2020), quando um código é escrito de forma limpa, seus leitores conseguem entender o que está lá especificado, reduzindo, assim, o custo de manutenção, facilitando a estimativa de tempo na implementação de outras funcionalidades, bem como a correção de erros, isto é, quanto mais fácil de entender, mais fácil é de se usar.

Outra especificação dada para a utilização de um código limpo é proposta:

> Deve-se buscar não utilizar repetições de código, para evitá-las crie funções, procurando não utilizar estruturas para tratar erros em funções específicas que chamem a função que tem relação com o funcionamento do software. As funções devem ter o mínimo de linhas possível, mas sem perder seu objetivo, usar o mínimo de parâmetros possível de preferência não ultrapassar três, procurar não criar aqueles enormes aninhamentos e deve ter uma endentação que o torne visualmente bonito. (MARTIN, 2009, p. 35)

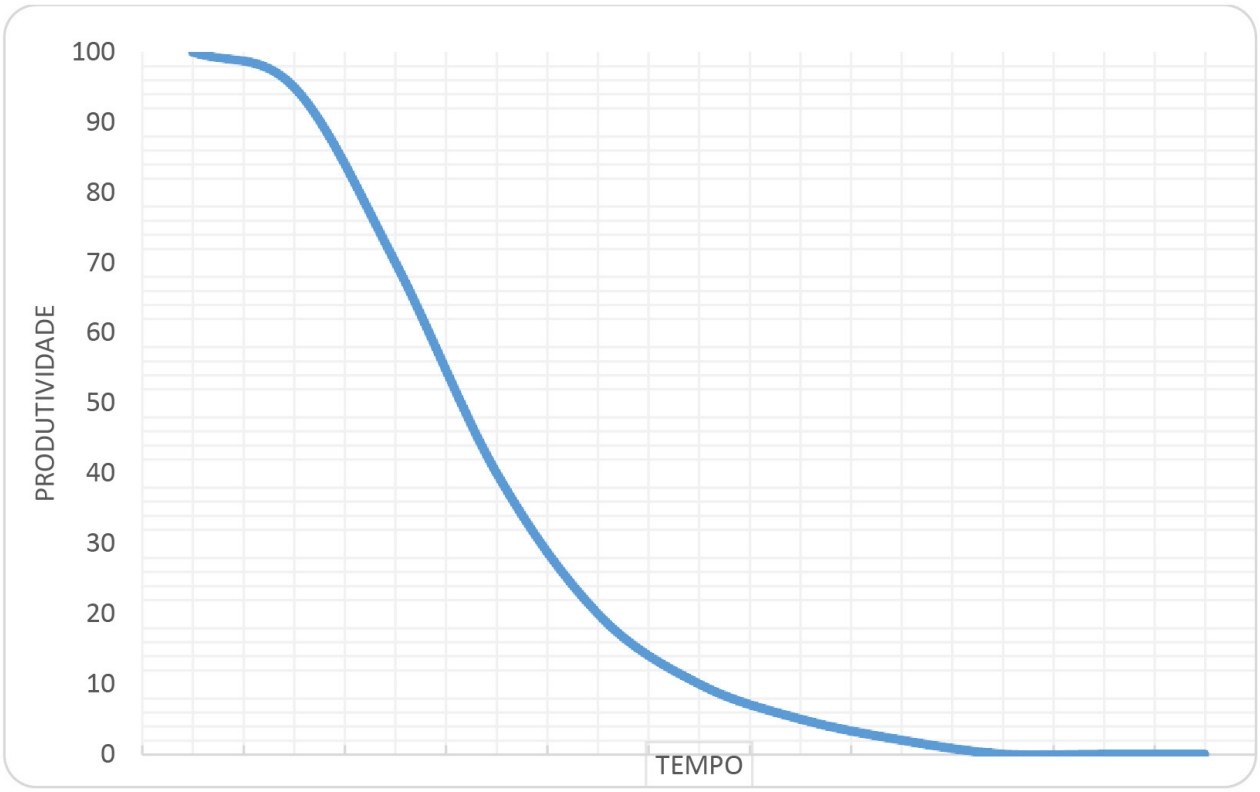
O autor ainda relata sobre a utilização, ao longo da escrita, do código de comentários como falta de condições de repassar informações acerca do

que o código realmente traz, isto é, tentam expressar o que cada trecho daquele código realiza, o que retrata uma programação sem nenhuma estrutura semântica.

Ao longo das rotinas, é necessário criar a confecção de códigos autoexplicáveis e seguir padrões, sem a utilização em massa de comentários, o próprio código deve ser detentor de sua da sua funcionalidade. O desenvolvedor deve ser capaz de criá-lo e manipulá-lo para que, futuramente, ele (ou outros profissionais) seja capaz de entendê-los de forma mais clara possível.

Lembre-se que conforme o código aumenta e se ele não possuir uma estrutura organizada, sua confusão tende a aumentar, diminuindo a produção por linhas de código. E quanto mais e mais esse emaranhado de código desordenado cresce, a sua redução de produtividade pode chegar a níveis insignificantes. Neste contexto, para tentar sanar esses problemas, as organizações tendem a adicionar mais programadores o que também é um erro, pois novos envolvidos no projeto requer tempo de treinamento para seu entendimento, alterando todo o cronograma a ser cumprido. O Gráfico 1 ilustra a produtividade de uma equipe de desenvolvimento.

Gráfico 1 – Produtividade versus Tempo de desenvolvimento



Fonte: adaptado de Martin (2019, p. 4).

Como observado no gráfico, a confusão dentro de um código-fonte pode ocasionar uma produtividade zero ou bem próxima a este valor, ou seja, o grau de desaceleração pode ser significativo. Cada mudança ou alteração sem planejamento pode levar à "morte" de um projeto de software, ou seja, ninguém da equipe consegue mais trabalhar naquele projeto. Deste modo, deverá ocorrer um replanejamento. Forma-se, então, uma nova equipe para recomeçar a construir um sistema do zero, mas já com tudo replanejado e com técnicas a serem seguidas para a criação de seu código-fonte e rotinas.

2. Código limpo

Mas, o que é um código limpo na versão dos desenvolvedores? Com certeza, a partir de suas experiências muitos desenvolvedores de

sistemas se acostumaram com sua própria “bagunça de código e até mesmo com a de seu colega de anos de trabalho em conjunto. Essa não é a maneira adequada de tratar sistemas computacionais, e os profissionais têm ciência disto, essa diretriz está profundamente errada, mesmo com tanta experiência pode acarretar perca de prazo e qualidade. Pois bem, a maneira mais sensata de trabalhar mais rápido e com mais clareza, segundo Martin (2019), é a utilização de um código limpo. Ainda de acordo com o autor, a definição de código limpo pode estar associada a inúmeros fatores que podem ser descritos de maneira bem particular por diferentes programadores, conforme descrito a seguir.

Bjarne Stroustrup, o criador do C++ (1983), descreve: o que caracteriza o seu código como limpo é um código elegante e eficiente, com lógica direta e livre de bugs, de fácil manutenção, com tratamento completo de erros, com eficiência para que não se torne um código confuso com otimizações sorrateiras de leitura natural.

Já Grady Booch descreve o código limpo como simples e direto, que atinja os objetivos do programador, com abstrações claras e linhas de controles objetivas e diretas.

Por fim, Dave Thomas, fundador da OTI, relatava que um código limpo é quando o desenvolvedor é capaz de ler seu código, executar testes de unidade e de aceitação, possuir nomes significativos, com poucas dependências, com API mínimo e claro, deve ser inteligível dependendo do tipo de linguagem a ser utilizada.

Diversos contextos norteiam o significado de código limpo, mas todos são capazes de mensurar que a sua escrita com qualidade e objetiva é fundamental para auxiliar o seu entendimento. Portanto, não basta apenas escrever códigos. Muitos programas funcionam perfeitamente com código ruins e com péssimas estruturas, também não basta

escrever um código apenas bom, ele deve ser mantido e possuir manutenibilidade sempre que necessário.

Algumas premissas básicas com relação a variáveis, funções, parâmetros, classes e arquivos fontes são interessantes de serem seguidos. Basta utilizar regras simples para deixar um código limpo e organizado, conforme mencionados a seguir.

2.1 Nomes

O nome deve levar o seu propósito, o que ele representa na realidade. Muitas vezes, escolher nomes pode levar tempo, mas este tempo será recompensado no futuro. Eventualmente, nomes podem sofrer alterações sempre que necessário, mas lembre-se bem: troque, mas troque por melhores, quando todos, inclusive o seu próprio criador quando ler poderá entender o que foi especificado.

O nome deve possuir um significado próprio, que relata a ação desenvolvida como descrito:

> Um nome deve poder ser lido na sua língua nativa, ou seja, nada de abreviaturas que criarão palavras inexistentes e sem nexo, não é necessário estabelecer abreviaturas de tipagem junto ao nome definido, pois o código deve ser simplificado, que não permitirão que seja encontrada com facilidade dentro do código, causando problemas nos momentos da manutenção dos programas. (MAYER; MAZER, 2015, p. 9)

Um nome, se não descrito claramente, simplesmente não significa nada. Ao lerem, deverão entendê-lo o porquê da sua criação. Observe dois códigos-fonte que descrevem um código PHP com dois nomes distintos para uma mesma função, conforme demonstrado:

```
<?php

    Function calcular (x, y) {
```

```
    Return x + y;

    }

?>
```

O código demonstra um código simples em linguagem PHP para uma função chamada calcular. Note que se trata de um cálculo matemático; mas, que cálculo é esse? Note o quão vago está o nome desta função; grosso modo, podemos dizer que não sabemos qual tipo de cálculo a função realiza. Agora, observe o próximo código.

```
<?php

    Function divisao (dividendo, divisor) {

    Return dividendo / divisor;

    }

?>
```

Com a fatoração é observado o tipo de cálculo que a função realiza. Trata-se de um cálculo matemático de divisão, e seu nome está explicito no código, suas variáveis expostas de forma clara, simples e objetiva.

Nomes de variáveis, classes ou funções devem estar convencidas a resolver questões que facilitem o entendimento de quem as irá utilizar, além de especificar a sua existência, possuir nomes significativos para que o usuário entenda sua funcionalidade. Uma variável qualquer que recebe como nome Int m, pode não expressar nada, não é capaz de indicar sua funcionalidade no código; não se sabe se retrata meses, idade em meses, pois simplesmente é o caractere m, solto e vago no código, ao contrário quando as variáveis são declaradas como a seguir:

- Int tempoDecorridoEmMeses
- Int idadeEmMeses,
- Int mesesDesdeModificacao

Observe que variáveis podem receber nomes maiores que os habituais, o que, de início, pode tornar mais difícil a sua criação, mas, ao final, é de fácil entendimento a todos que acaso irão utilizar ou modificar o código.

Magalhães e Tiosso (2019) ressaltam que apesar de ser banalizados, os comentários são necessários, desde que sejam breves. Mas, primeiramente, deve-se observar que não possui alguma maneira de expressar as informações apenas via código-fonte. A Figura 2 ilustra comentários feitos de maneira que auxiliem o entendimento do método o entendimento das funcionalidades dos métodos.

Figura 2 – Lista de comentários em um sistema computacional

```
// Retorna a lista de produtos vendidos
// para o relatório de fechamento mensal
public ListaCompras<Produto> ObtemProdutosVendidos()
{
    ...
}
```

Fonte: elaborada pela autora.

3.2 Classes

Devem ser representadas por substantivos simples (Cliente, Estoque, Setor), e jamais verbos; devem possuir responsabilidade única e estar coesa com o que o código trata, por exemplo:

// Evite: Class DadosAluno

// Utilize: Class Aluno

Ao utilizar apenas o substantivo Aluno, está sendo referenciado todas as informações sobre o aluno, todas as ações (métodos) que serão tratadas dentro daquela classe e não apenas seus dados, ou seja, poderão ser realizadas ações de cadastrar, login, listar, entre outras.

3.3 Funções

Simples, claras e objetivas, de fácil entendimento e, principalmente, pequenas. Cada uma das funções criadas deve especificar seu papel de apenas e exclusivamente uma única coisa, sempre as nomeie por verbos, pois são elas as responsáveis pela execução do sistema. Funções não devem possuir argumentos, mas caso isso não seja possível ou que possua uma quantidade maior que 3 de argumentos; é recomendado a criação de uma classe própria, como no exemplo.

// Evite: Cadastrar (nome, sobrenome, RG, CPF, sexo, dataDeNascimento)

// Utilize: Cadastrar (DadosPessoais dadospessoais)

Ao utilizar atributos DadosPessoais, este engloba todos os dados pessoais que deseja se cadastrar e não apenas como já mostrado de uma quantidade limitada (nome, sobrenome, RG, CPF, sexo, dataDeNascimento).

Outro exemplo de declaração de funções:

// Evite: function dadosE($string)

// Utilize: function obterDadosClientePorCPF($CPFCliente)

Na segunda opção é fácil perceber que no momento que for fornecido o CPF do cliente, obtém-se como resposta seus dados.

Outra questão importante é a declaração de funções de acordo com a sua grandeza, ou seja, é importante para um bom entendimento do

código começar a lê-lo de cima pra baixo, sempre funções menores e, posteriormente, declarando as maiores, em um nível descendente de abstração conforme. Assim:

```
// Utilize

public void CadastrarAluno(string nome) { ... }

public void CadastrarCliente(string nome, int idade) { ... }

public void CadastrarCliente(string name, int idade, int
RegistroAcademico) { ... }
```

3.4 Variáveis

As variáveis devem possuir, também, nomes significativos, com uma localização próxima de onde serão utilizadas; evite criá-las juntas uma abaixo da outra.

```
// evite: var total = 0;

public void CadastrarCliente() { ... }

public void CadastrarPedido() { ... }

public void AtulizarPedido() { ... }

public void CalcularTotal()

{

    total = 30;

}
```

Observe que existe um longo caminho a ser percorrido até onde a variável total realmente irá ser utilizada, isso dificulta a vida do programador, que perde tempo em procurar onde e como esta variável foi declarada.

```
// utilize

public void CadastrarCliente() { ... }

public void CadastrarPedido() { ... }

public void AtualizarPedido() { ... }

var total = 0;

public void CalcularTotal()

{

    total = 30;

}
```

Neste momento é perceptível aos olhos de quem vai utilizar o **código** que a variável total corresponde ao valor atribuído a ela dentro do método CalcularTotal, ao abri-lo, estará bem à frente de quem irá utilizá-la.

3.5 Métodos

Devem ser declarados em forma de verbos, quanto menor possível mais fácil de gerenciar, crie com nomes curtos. É mais fácil interpretar métodos menores e reutilizáveis do que tudo dentro de um método grande de difícil leitura e compreensão, conforme:

```
// Evite: public void RealizarPedido()

// Utilize:

public void CadastrarCliente() { ... }

public void AplicarDesconto() { ... }

public void AtualizarDesconto() { ... }

public void SalvarPedido() { ... }
```

A primeira opção deve ser evitada, pois engloba várias ações de um método grande, o ideal é condensá-los em pequenos blocos de métodos, ou seja, vários métodos.

3.6 Comentários

São extremamente úteis quando bem colocados, mas comentários em código ruins, pode ser um desastre, disseminando informações imprecisas que não trazem nenhum tipo de apoio. Devemos utilizá-los sempre que a escrita do código não consiga expressar o que ele realmente faz. Lembrem-se que os códigos sofrem alterações, são modificados sempre que necessário e evoluem. Além disso, o programador deve ser disciplinado e não se esquecer de atualizar os comentários, estabelecendo a sua nova especificação. Códigos devem ser claros; evite comentários sem contexto que não fazem sentido algum ao cenário daquele bloco de código.

```
// Evite

public void FuncaoNumeroInteiro()

{
```

```
    // string texto = “1234”;

    // public void Metodo() {... }

}

// Utilize

// Retorna a lista de produtos inativos

// para o relatório de fechamento mensal

public IList<Product> ObtemProdutosInativos()

{ }
```

Observe que no primeiro exemplo o comentário nada tem a dizer, pois não explica o que está acontecendo na função ****. Já o próximo exemplo, explica de forma muito clara a funcionalidade do método, ou seja, o que ele realmente está propondo. Vejamos:

```
// Utilize

public void CancelarCompra()

{

    // Caso a compra já tenha sido enviada

    // ela não pode mais ser cancelada.

    if(DataEnvio > DateTime.Now)

    {

    AddNotification(“Compra realizada com sucesso, impossível
    cancelamento”);
```

```
    }

}
```

Neste exemplo, o comentário, de forma clara e objetiva, relata ao desenvolvedor que o método é responsável pelo cancelamento de um produto. Se a compra já tenha sido enviada para o cliente, será impossível realizar o seu cancelamento.

Portanto, utilize comentários que informe uma regra de negócio ou funcionalidade; evite usar palavras reservadas ao código, como nome de funções, variáveis ou métodos, o melhor é informar o que aquele trecho de código executa.

3.7 Nomes significativos e pronunciáveis

Como dito, os nomes devem refletir a sua finalidade. Escrever bons nomes que expliquem verdadeiramente o código é essencial para um código limpo e entendível.

```
// Evite: var a = 21;

// Tempo de quê? Qual parâmetro de medida?

int tempo = 12
```

Percebam que “a” não significa nada; pode ser ano, altura ou qualquer outro valor recebido. O mesmo acontece com a variável “tempo”, em que este tempo refere-se a quê? A quantidade de tempo em segundos, minutos, anos? Ninguém sabe dizer, talvez nem a própria pessoa que o criou.

```
// Utilize

intTempoMeses = 12
```

Deste modo, está bem mais claro que o tempo é decorrido em meses. O programador, ao abrir o código, terá a noção explicita de que a variável trata de meses e não dias ou anos.

// Evite: Var salario = 1100;

// Utilize: Var salarioEmReais = 1100;

Quanto mais específico forem os nomes, maior é a chance de os desenvolvedores entenderem o código e mais rápido poderão ser feitas as alterações precisas que não alterem a estrutura nem a qualidade do sistema.

Os nomes também devem ser pronunciáveis e passíveis de busca, evite nomes com uma letra apenas, ou conjunto de números, sem codificação.

Evite nomes sequenciáveis: a1, a2, a3 e a4; não são capazes de expressar alguma informação sobre o que realmente querem transmitir. O leitor do código deve compreender as diferenças presentes em cada um dos nomes, exigindo por parte do desenvolvedor um gasto enorme em mapa mental, isto é, tempo perdido tentando decifrar o nome escolhido por ele mesmo ou por outros desenvolvedores.

As codificações também devem ser evitadas:

gr2021( ) → Gera relatório

Difícil de entender que “gr” significa gerar relatório. Por isso, escreva de maneira mais limpa, conforme a seguir:

// Utilize: GerarRelatorios2021 ( ) → Gerar relatório

É inaceitável o uso de gírias ou coloquialismo. É imprescindível ser profissional, técnico, formal; use nome de domínios quando for preciso.

Os desenvolvedores estão habituados a padrões de algoritmos quando for a hora de aplicá-los.

Referências

LEWIS, Elijah. **Clean Code:** Advanced and Effective Strategies To Use Clean Code Methods. São Paulo, 2020.

MAGALHÃES, Phelipe Al.; TIOSSO, Fernando. **Código Limpo:** padrões e técnicas no desenvolvimento de software. Revista Interface Tecnológica, v. 16, n. 1, p. 197-207, 2019. Disponível em: https://revista.fatectq.edu.br/index.php/interfacetecnologica/article/view/597/348. Acesso: 25 maio 2021.

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

MAYER, Carlos R.; MAZER, Ademir Jr. **Visão Introdutória sobre os conceitos do código limpo.** 2015. Disponível em: https://semanaacademica.org.br/system/files/artigos/visao_introdutoria_sobre_os_conceitos_de_codigo_limpo.pdf. Acesso em: 1 out. 2021.

Técnicas de usabilidade: melhorando a qualidade da experiência do usuário

Autoria: Stella Marys Dornelas Lamounier
Leitura crítica: Marco Ikuro Hisatomi

Objetivos

- Aprimorar técnicas de usabilidade em projeto de software.
- Demonstrar métricas de usabilidade.
- Abordar o conceito de experiência do usuário em sistemas computacionais.
- Apresentar as principais ferramentas para a medição da experiência do usuário.

1. Usabilidade

A evolução computacional, o advento da internet e a competitividade no desenvolvimento de software fizeram com que desenvolvedores de sistemas aprimorassem suas técnicas do processo de desenvolvimento, seja ela em criação de código-fonte mais organizados e com mínimo de erros possíveis, e também partes que englobam análise e levantamento de requisitos, a fim de englobar o software como um todo.

Neste contexto, as empresas de desenvolvimento começaram a se preocupar em garantir uma melhor aceitação do software a partir de técnicas que garantissem melhor facilidade de uso, agilidade na navegação, redução de riscos/erros e minimização de problemas ligados a usabilidade.

A usabilidade está amplamente ligada à facilidade, eficiência e efetividade de um produto, uma área que também cresce de acordo com a complexidade do software. Deve estar presente em todo o processo de criação do sistema computacional, sendo parte da vida de quem programa e cria sistemas voltados para a qualidade e satisfação do usuário.

A ISO 9241-11 (ABNT, 2002) define o termo como um fator de qualidade base para garantir produtividade e segurança em um contexto de uso bastante específico. A usabilidade pode ser aplicada nos mais diversos tipos de produtos, mas destacaremos no contexto computacional como:

> É tornar o código o mais útil e fácil de usar. «Usabilidade não é apenas atendimento aos requisitos do usuário; trata-se de criar experiências que permitam ao usuário atingir seus objetivos com o mínimo de aborrecimento, tempo e esforço cognitivo. (PADOLSEY, 2020, [s.p.])

Seguindo estes conceitos, pode-se dizer que o termo é independente da sua área de aplicação e está fortemente ligado à facilidade de uso,

prevenção/redução de erros, satisfação do usuário, produtividade e execução de tarefas. Deve ser de interesse de todos, pois trata-se de um processo colaborativo, interdisciplinar; assim, é importante que todo membro da equipe envolvida tenha consciência da necessidade da usabilidade e conheça o básico da sua aplicação.

A mesma norma ISO em seu contexto geral destaca três premissas para detalhar o termo usabilidade em computadores: eficácia, satisfação e eficiência conforme ilustradas na Figura 1:

Figura 1 – Usabilidade Padrão ISO 9241-11

Fonte: adaptada de ABNT (2002, [s.p.]).

Sendo elemento central, a usabilidade, na produção de softwares mais fáceis e produtivos, onde a medidas de eficácia estão relacionadas aos

objetivos do usuário quanto à sua acurácia e completude com que estes objetivos podem ser alcançados. Enquanto a eficiência está relacionada a recursos gastos em relação à acurácia e abrangência com as quais usuários atingem objetivos. E é pela satisfação se mede a extensão pela qual os usuários estão livres de desconforto e suas atitudes em relação ao uso do produto.

É notável a utilização dessa premissa nas mais diversas partes do ciclo de vida do software, por exemplo, na codificação da aplicação, criação de funções, desenho das páginas quando se tratar de sistemas web ou na arquitetura. Com isso, a responsabilidade é demonstrar ao usuário que utiliza o software uma forma de encontrar o que está buscando e maneira simples e rápida, sentindo-se à vontade sempre que quiser ou quando precisar.

A seguir, serão abordadas algumas técnicas para satisfazer a interação do usuário com o projeto de software.

1.1 Técnicas de usabilidade

Diversas são as técnicas para aumentar a usabilidade em sistemas computacionais. Ter um sistema de fácil utilização sem dúvida é sinal de aumento de produtividade por parte do usuário e garantia de qualidade por parte dos desenvolvedores de software.

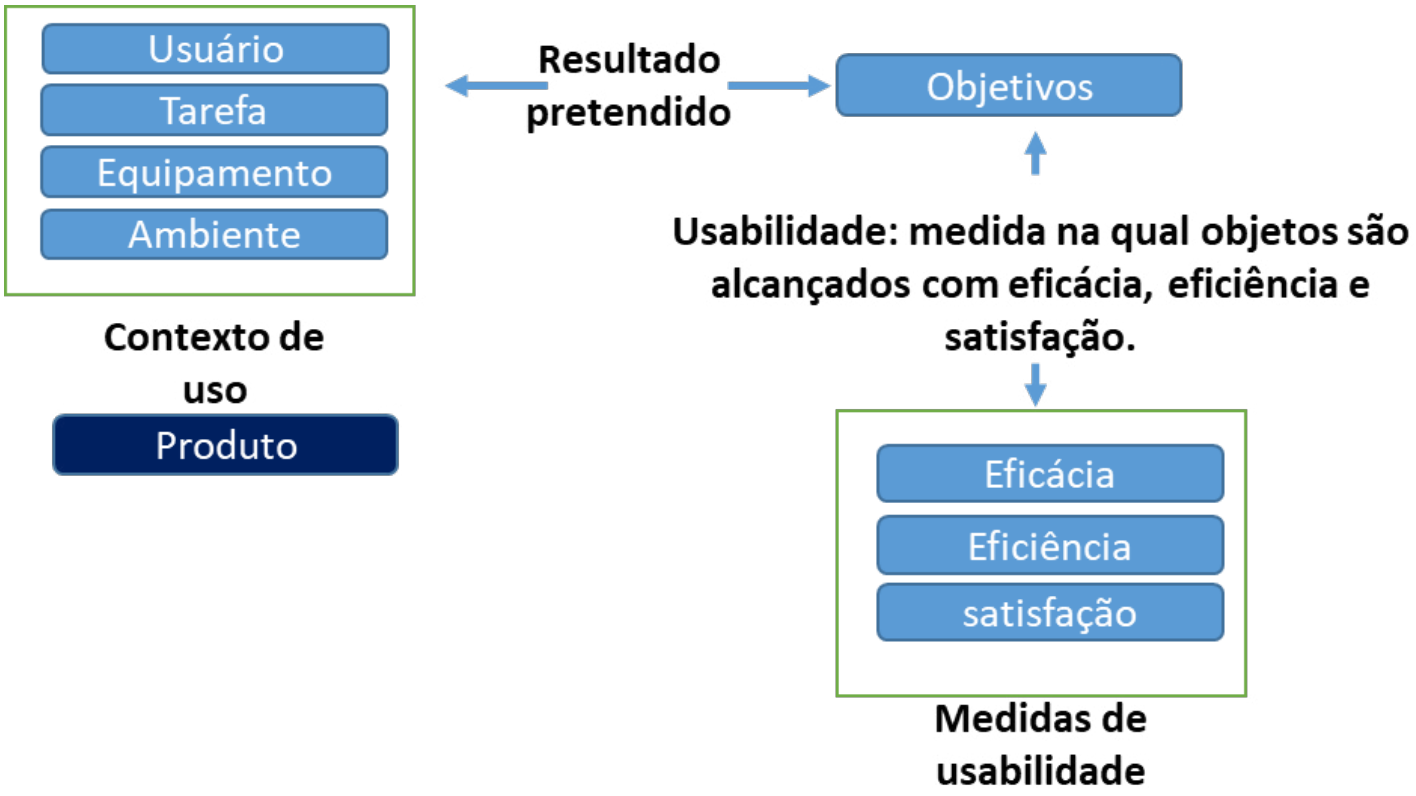
Para adotar uma técnica adequada deve-se considerar a metodologia de desenvolvimento da empresa, isto é, somente as pessoas envolvidas na criação do projeto serão capazes de aplicar modelos que, ao final, irão garantir a satisfação do usuário final.

Estas técnicas devem ser usadas como ferramentas de apoio à medição de usabilidade, como mecanismo de melhoramento do sistema. Os critérios de medidas irão depender dos requisitos e da necessidade da organização, que podem dividi-los de acordo com critérios mínimos de

aceitação especificado em porcentagem ou em valores de referências. Caso esteja medindo o tempo para executar uma determinada tarefa, por exemplo: 30 segundos ou que não pode exceder tempo máximo de 1 minuto, ou para um montante de cinco usuários que podem completar determinada tarefa em no máximo 2 minutos – a depender da regra de negócio adotada pela empresa.

Porém, antes de descrevermos algumas técnicas de medidas de usabilidade é preciso identificar os objetivos que norteiam o sistema, ou seja, o que se pretende com a decomposição dos três requisitos básicos (eficiência, eficácia e satisfação), conforme ilustrado na Figura 2, para posteriormente descrever os objetivos pretendidos, componentes de uso do ponto de vista do usuário, de tarefas, equipamentos e ambientes. E principalmente quais são os valores reais ou desejados para medir a eficiência, eficácia e satisfação do produto de software.

Figura 2 – Objetivos que norteiam o sistema



Fonte: adaptada de ABNT (2002, p. 4).

Para o detalhamento do nível de cada uma destas medidas deve-se levar em consideração os objetivos das partes envolvidas na medição, a importância de cada medida para cada um destes objetivos que podem ser do tipo objetivo global ou objetivos menores. O Quadro 1 ilustra de forma prática os objetivos globais.

Quadro 1 – Medida de usabilidade

Objetivo global	Medidas de eficácia	Medidas de eficiência	Medidas de satisfação
Objetivos de usabilidade	• Porcentagem de objetivos alcançados. • Porcentagem de usuários completando a tarefa com sucesso. • Média da acurácia de tarefas completadas.	• Tempo para completar uma tarefa. • Tarefas completadas por unidade de tempo. • Custo monetário de realização da tarefa.	• Escala de satisfação. • Frequência de uso. • Frequência de reclamações.

Fonte: adaptado de ABNT (2002, p. 4).

Após um breve contexto sobre os objetivos globais, serão ressaltadas algumas técnicas comuns para medir a usabilidade em produto de software. A técnica de testes de usabilidade, por exemplo, é enquadrada nos modelos de “caixa-preta”, isto é, a responsabilidade neste modelo é jamais se preocupar com questões relacionadas à parte interna do sistema, com código-fonte, linguagem de programação ou banco de

dados. Essa técnica atenta para questões ligadas às funcionalidades e à satisfação do usuário a partir de levantamento de requisitos, características e comportamentos esperados, sempre buscando descobrir as falácias, pontos a serem melhorados no projeto, como medição de desempenho e precisão.

Utilizar-se de métricas de testes é importante para diagnosticar qual parte do sistema o usuário possui dificuldades em realizar suas tarefas. Portanto, avaliando se o usuário consegue concluir com sucesso alguma tarefa ou rotina e quantos usuários conseguem concluí-las sem cometer um erro sequer.

Segundo Guimarães (2019), existe uma métrica bastante usada para mensurar desempenho, conhecida como “taxa de conclusão da tarefa” que mede a porcentagem das tarefas executadas corretamente pelos usuários.

$$Efetividade = \frac{\text{Total de tarefas concluídas com sucesso}}{\text{Número total deTotal de tarefas}} x\ 100\%$$

Aplicando a equação de efetividade, temos um total de tarefas concluídas com sucesso = 7 e como número total de tarefas 10. Deste modo:

$$Efetividade = \frac{7}{10}\ x100\% = 70\%$$

Podemos notar que em um total de 10 tarefas temos 7 que foram concluídas com sucesso. Assim, entende-se que houve maior facilidade de concluir tarefas.

Lembre-se que é necessário saber definir o que significa sucesso para um software, como: O sucesso é finalizar uma compra? Emitir relatório? Entrar em um site e conseguir realizar uma compra? Tudo isso deve ser

levado em consideração, pois, quanto maior a taxa de sucesso de uma tarefa melhor, isto significa que os usuários estão conseguindo interagir muito bem com o sistema.

Outra métrica é o tempo de execução que pode ser medido por minutos, segundos etc. Ela analisa quanto tempo um certo usuário leva para concluir uma determinada tarefa, onde é feita a média da soma de todo o tempo gasto do conjunto de tarefas realizadas por todos os usuários, como demonstrado no Quadro 2:

Quadro 2 – Tarefas dos usuários

Tarefa	Usuário	Tempo (segundos)
Finalizar compra	Maria	20
	Antônio	36
	Pedro	12
	Joana	40
	Carlos	20

Fonte: elaborado pela autora.

Aplicando o cálculo matemático:

$$Média = \frac{20 + 36 + 12 + 40 + 20}{5} = \frac{128}{5} = 25{,}6\ segundos$$

Realizando a média simples do tempo de execução da tarefa é perceptível que alguns usuários possuem mais dificuldades que outros, mas o importante é descobrir qual tempo de navegabilidade para execução da tarefa. Caso o tempo for relativamente lento, deve haver uma formulação na funcionalidade, ou seja, quanto menor o tempo gasto na tarefa, melhor é a experiência do usuário.

Quando se trata de tempo de realização de tarefa, deve-se levar em consideração o tempo que o usuário leva para realizar uma atividade pela primeira vez e, posteriormente, pela segunda; identificar se há um aprendizado em relação ao uso do sistema, assim como se a pessoa possui algum domínio com a tecnologia empregada.

Outra técnica simples e eficaz utilizada como métrica de usabilidade para medir a satisfação do usuário é a aplicação de questionário, que pode ser aplicado tanto no final de cada tarefa quanto durante a rodada de testes. Esses questionários devem ser compostos por questões que possam ser capazes de medir o grau de dificuldade para a realização de tarefas, mesmo que de forma subjetiva. Observe alguns exemplos a seguir.

Lembre-se que estes questionários e sua aplicação estarão vinculados à metodologia de trabalho da empresa que os aplica. É de sua responsabilidade saber diferenciar os tipos que mais se adaptam àquele projeto e quais que podem trazer os resultados mais promissores.

A empresa precisa ter em mente o perfil dos participantes que irão responder a estes questionários, qual seu conhecimento tecnológico quando se trata da usabilidade computacional. É importante procurar pessoas de regiões demográficas diferentes, com nível de acesso a tecnologias, além de tentar fomentar pessoas que não sejam apenas do conhecimento ou convívio da equipe. Por fim, para ter sucesso nos testes, não é necessário ter um grande número de amostras. O ideal é saber planejar o perfil e o público-alvo que deverá ser atingindo com o projeto. Não existe um número certo, engessado, dependendo da complexidade do sistema; apenas a equipe será capaz de adicionar ou excluir o número de participantes, até que se chegue em uma quantidade considerada ótima para realizar os testes de usabilidade de forma que não atrapalhe a entrega o software.

Braum (2019) descreve o questionário proposto por Likert (1932) que deve ser aplicado em testes de usabilidade, em que o convidado deverá respondê-lo a fim de emitir o seu grau de concordância naquela frase.

1. Eu usaria este sistema com frequência.
2. Esse sistema é desnecessariamente complexo.
3. O sistema é fácil de usar.
4. Eu precisaria de ajuda de uma pessoa com conhecimentos técnicos para usar o sistema.
5. As funções do sistema são muito bem integradas.
6. O sistema apresenta muita inconsistência.
7. As pessoas aprenderão como usar esse sistema rapidamente.
8. O sistema é confuso para usar.
9. Senti confiança ao utilizar o sistema.
10. Precisei aprender várias coisas novas antes de conseguir utilizar o sistema.

Todas as perguntas são compostas por cinco respostas ilustradas como exemplo fictício no Quadro 3, que podem sofrer alterações no seu conjunto de afirmações, mas não deve jamais alterar a ordem cronológica das questões, pois podem comprometer o resultado esperado.

Quadro 3 – Escala de Likert

Estou satisfeito com o serviço recebido				
Discordo totalmente	Discordo parcialmente	Não concordo nem discordo	Concordo parcialmente	Concordo totalmente
1	2	3	4	4

Fonte: Júnior e Costa (2014, p. 4).

A título de exemplo, iremos aplicar um cálculo matemático para a concretização do resultado final obedecendo aos critérios propostas pela Escala de Usabilidade (SUS):

Quando se tratar de questões com números ímpares (1, 3, 5, 7, 9) seu *score* é obtido na escala subtraindo -1.

Quando se tratar de questões com número pares (2, 4, 6, 8, 10) seu *score* é obtido na escala subtraindo 5.

A soma destes *scores* é multiplicada pela constante 2,5. O resultado deverá estar dentro de uma faixa de valor que vai de 0 a 100, isto é, não poderá haver resultados negativos ou superiores a 100.

Vejamos como fica o cálculo para a usabilidade de uma tela de cadastro de alunos ilustrado pela Tabela 1.

Tabela 1 – Tabela para cálculo de Fator de Escala – Usabilidade

Questão	Fator de escala
1	5. Concordo totalmente.
2	2. Discordo.
3	1. Discordo totalmente.
4	3. Neutro.
5	4. Concordo.
6	5. Concordo totalmente.
7	3. Neutro.
8	1. Discordo totalmente.
9	3. Neutro.
10	2. Discordo.

Fonte: elaborada pela autora.

Cálculo:

- Questões Ímpares = (5-1) + (1-1) + (4-1) + (3-1) + (3-1) = 11.
- Questões pares = (5-2) + (5-3) + (5-5) + (5-1) + (5-2) = 12.
- Total geral = (11 + 12) x 2,5 = **57,5%.**

A avaliação da pontuação pode ser medida de acordo com o *score* obtido, neste caso, o valor de *score.* Ou seja, com o resultado final superior a 68 pontos, o produto ou software apresenta vários problemas a respeito da usabilidade. Mas lembre-se que este foi apenas um exemplo. É adequado realizar este teste em uma amostragem significativa de usuários, mas deve-se mensurar a escala completa de pontuação proposta por Likert (1932).

- Menor que 60%–inaceitável.
- Maior que 60% e menor 70%–OK.
- Maior que 70% e menor que 80%–Bom.
- Maior que 80% e menor que 90%–Excelente.
- Maior que 90%–A melhor usabilidade possível.

Existem outras métricas para se medir a usabilidade, porém, depende de inúmeros fatores, como: a experiência dos usuários; perfil da empresa; ou modelo de aplicação que está sendo testado. O importante é priorizar o projeto, deixando-o mais usual possível, com maior facilidade de uso para que o usuário tenha o conforto necessário para realizar suas tarefas.

2. Experiência do usuário – *User Experience*

Experiência do usuário, termo também conhecido como UX, em sistemas computacionais, tornou-se um emblema muito importante, pois é quem trata o planejamento e concepções de experiência que o usuário estabelece quando interage com o software. Vale lembrar que este termo também pode ser amplamente empregado não só apenas em sistemas, mas em diversos produtos como eletrônicos e eletrodomésticos.

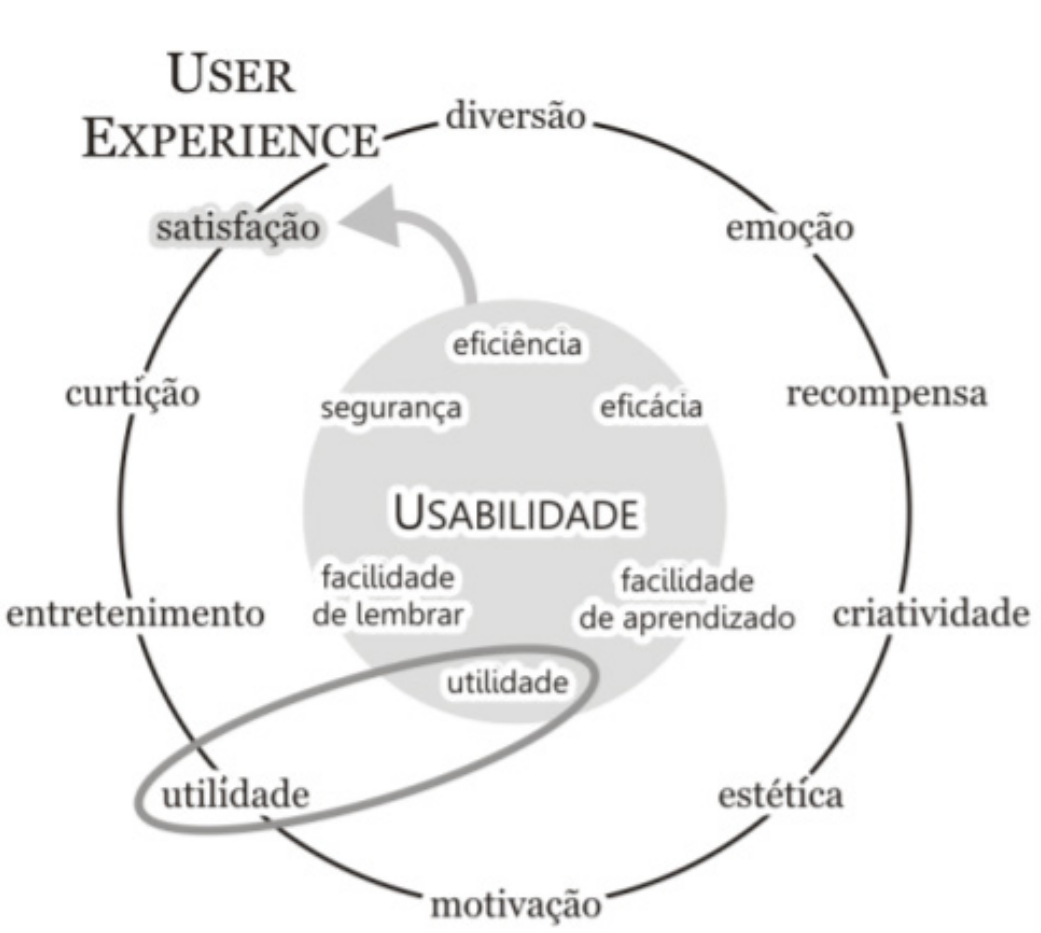
Avaliar produtos ou softwares a partir da experiência do usuário, contribui para que sejam projetos capazes de atender e auxiliar os usuários na realização de suas tarefas do dia a dia de forma mais fácil e harmoniosa.

A usabilidade em seu contexto geral, tem forte relação com a UX, pois ambas estão preocupadas com a percepção do usuário, sua facilidade de interação com o sistema.

A experiência do usuário também refere-se a um tema um pouco subjetivo, que pode ser interpretada com perspectivas diferentes, mas com objetivos comuns definidos pela Associação Brasileira de Normas Técnicas (ABNT): “Experiência do usuário: percepções e respostas das pessoas, resultantes do uso e/ou uso antecipado de um produto, sistema ou serviço (ABNT, 2011).

Pode-se dizer que a UX nada mais é que a evolução da usabilidade, pois, além das características que a acompanham, deve ser norteada por aspectos que integram a estética. Sensações e emoções devem estar presentes ao lidar com o produto a ser utilizado. A Figura 3 ilustra as principais diferenças entre os dois termos.

Figura 3 – Usabilidade e *User Experiencie* (Experiência com usuário)



Fonte: Padovanil, Schlemmer e Scariot (2012, p. 4).

Apesar de se mostrarem similares, possuem diversas características diferentes. A UX está mais preocupada com motivação, estética, criatividade, recompensas, diversão, satisfação e entretenimento, enquanto a usabilidade está ligada a fatores como eficiência, segurança, eficácia e facilidade. Atente ao quão importante a experiência é a nível de usuário, pois, é por meio dela que se chega à resposta sobre a funcionalidade do sistema.

Segundo Morville (2004), ao se falar em experiência do usuário em sistemas computacionais é imprescindível que o termo deva possuir características básicas capazes de contemplar o usuário. Seu contexto e conteúdo ilustra os aspectos que devem estar presentes na UX, conforme a Figura 4.

Figura 4 – Componentes UX

Fonte: adaptada de Matos (2021, p. 17).

A funcionalidade de cada componente UX:

- Útil – O produto deve ser útil tanto do ponto de vista do usuário quanto para a empresa que o desenvolveu. Deve-se medir o grau de utilidade que o caracteriza.

- Desejável – A busca por um produto eficiente deve estar atrelada a elementos subjetivos que envolvem aspectos emocionais, como valor de imagem, do usuário, da organização e do produto.

- Acessível – Oferecer a possibilidade a qualquer usuário de acessar o sistema é de suma importância, pois este requisito permite a todos os usuários utilizarem o sistema facilmente, sem barreiras que possam atrasar a utilização do software.

- Creditável – O sistema deve transmitir confiança e credibilidade para quem irá utilizá-lo, tanto em relação ao seu designer quanto ao conteúdo apresentado.

- Localizável – O projeto de software deve levar o usuário a localizar com facilidade o que precisa, a fim de atingir seus objetivos o mais rápido e amigável possível.

- Usável – Juntamente com a usabilidade, o sistema deve ser usual; a facilidade de uso deve estar presente em todo o contexto do projeto.

- Valor – Devem oferecer valor de negócio às empresas que o desenvolvem e, também, às pessoas que o utilizam a partir de sua satisfação com o projeto adquirido.

Neste contexto, é sabido que a UX deve estar apta a trazer benefícios ao software e que proporcione um volume de tráfego maior, ou seja, capaz de atrair mais cliente ou números de navegantes quando se tratar de website com o mínimo ou nenhum obstáculo ao utilizá-lo, a fim de completar a tarefa iniciada com poucos reparos e manutenção.

2.1 Técnicas para experiência do usuário

Por onde começar a aplicar técnicas que podem avaliar a experiência do usuário para um produto ou projeto de software? Existem inúmeros métodos que podem medir estes aspectos abordados em alguns modelos amplamente empregados dentro das empresas. Lembre-se que a experiência do usuário não é baseada apenas em telas; deverá ser capaz de estudar o comportamento do usuário a partir do serviço oferecido a fim de aprimorar sua experiência.

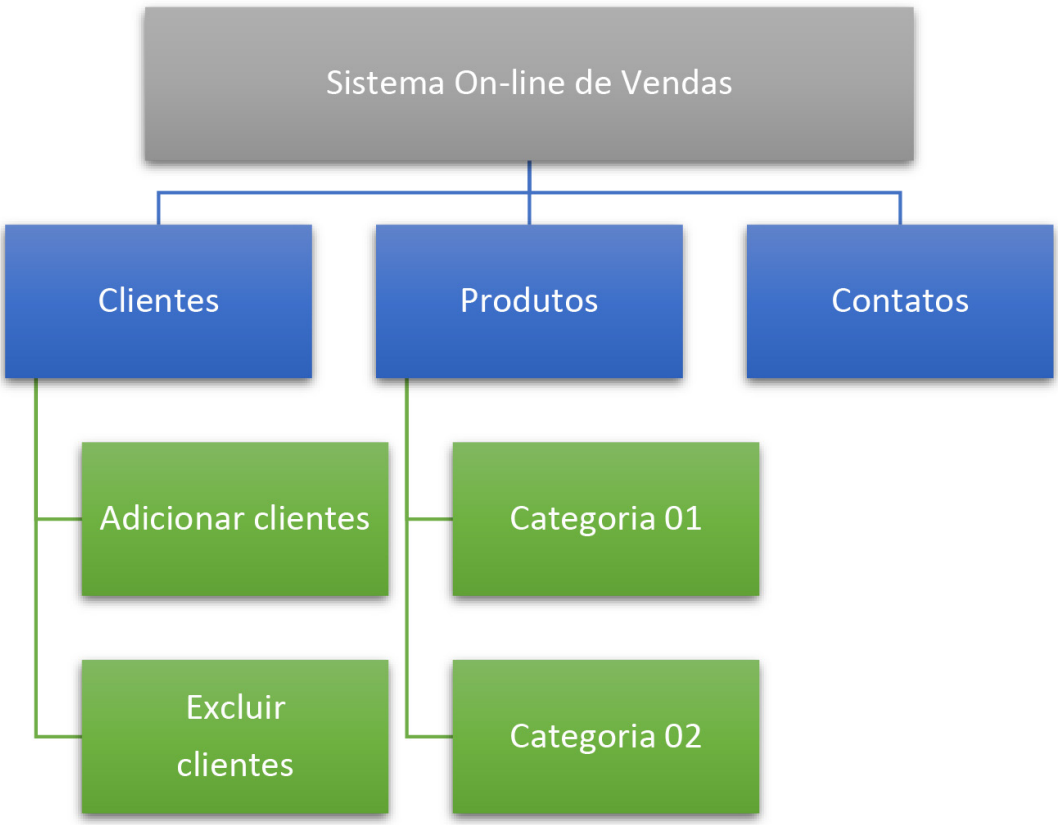
***Wireframes* ou estrutura de arame** é utilizada para uma organização visual, a depender do tipo de sistema a ser criado, como também

a experiência que o professional tem com esse tipo de ferramenta. Pode-se dizer que é um dos modos mais primitivos de visualizar o projeto, pois trata-se de um esboço inicial em que estão representadas visualmente as seções e diagramações de um projeto. É, ainda, um esboço criado pelos profissionais antes mesmo da criação de um layout para facilitar a visualização e o ajuste do produto. Podemos citar o Lucidchart e Mockingbird como ferramentas para a criação de Wireframes.

Protótipos: são modelos preliminares do sistema para que o cliente possa ter uma visão mais prática do produto que ele está adquirindo. Tem como função servir de modelo ou exemplo de um determinado produto. Sendo a versão inicial em funcionamento que pode servir tanto para testes quanto para interação com o usuário, como ferramenta para criação de protótipos e, também, para Wireframes Pencil Project.

Sitemap (mapa do site): é responsável por garantir o fluxo do sistema, fornecendo informações sobre os caminhos da aplicação ou de um site. Por exemplo, o Google utiliza esse mecanismo para rastrear um site e a relação do usuário com ele de forma inteligente. Deste modo, sua função é mapear a estrutura de um site e o que está incluído dentro dele de forma mais eficaz, por meio de mapas de navegação, fluxogramas, testes de usabilidade, mensuração de resultados e, claro, muita pesquisa. A Figura 5 ilustra um exemplo de Sitemap.

Figura 5 – Sitemap para *e-commerce*



Fonte: elaborada pela autora.

Existem outras ferramentas e técnicas no mercado para avaliar a experiência do usuário, como a análise retrospectiva, a experiência por amostragem, pesquisa e validação. *Brainstorming* é muito utilizada para a resolução de problemas por meio da criação e desenvolvimento de novas ideias. As personas representam um tipo de cliente, retratando o público-alvo a partir do comportamento, localização e necessidade trata de pessoas pertencentes a esses públicos que interagem com os desenvolvedores durante o processo de criação de designer.

Portanto, ferramentas existem, basta saber escolher a que mais se adapta ao seu perfil e, também, levar em consideração a experiência do professional, na busca por criar sistemas cada vez mais usuais e de fácil manipulação para o usuário final.

Referências

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **ABNT NBR ISO 9241-210**. 2011. Ergonomia da interação humano-sistema. Disponível em: https://www.abntcatalogo.com.br/norma.aspx?ID=088057 . Acesso em: 9 jun. 2021.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **ABNT NBR ISO 9241-11**. 2002. Requisitos Ergonômicos para Trabalho de Escritórios com Computadores Parte 11 – Orientações sobre Usabilidade. Disponível em: https://www.inf.ufsc.br/~edla.ramos/ine5624/_Walter/Normas/Parte%2011/iso9241-11F2.pdf. Acesso em: 9 jun. 2021.

BRAUM, Marianne. Guia: Como medir a usabilidade de produtos com System Usability Scale (SUS). **UX Collective,** [s.l.], 26 de novembro de 2017. Disponível em: https://brasil.uxdesign.cc/guia-como-medir-a-usabilidade-de-produtos-com-system-usability-scale-sus-e08f4361d9db. Acesso em: 9 jun. 2021.

GUIMARÃES, W. 2019. Métricas em testes de usabilidade, como usá-las para melhorar o seu produto. **UX Collective**, [s.l.], 21 de abril de 2019. Disponível em: https://brasil.uxdesign.cc/m%C3%A9tricas-em-testes-de-usabilidade-como-us%C3%A1-las-para-melhorar-o-seu-produto-parte-i-a275227240df Acesso em: 30 set. 2021.

JÚNIOR, Severino D. da S.; COSTA, Francisco J. Mensuração e escalas de verificação: uma análise comparativa das escalas de Likert e Phrase Completion. **PMKT–Revista Brasileira de Pesquisas de Marketing, Opinião e Mídia**, v. 15, n. 1-16, p. 61, 2014. Disponível em: http://sistema.semead.com.br/17semead/resultado/trabalhospdf/1012.pdf. Acesso 08 mai 2021.

LIKERT, R. A technique for the measurement of attitudes. **Archives of Psychology**, [s.l.], n. 140, p. 44-53, 1932.

MACEDO, Vanessa D. de. **Métodos de avaliação da Experiência do Usuário (UX) com eletrodomésticos**: um estudo exploratório. 2014. 144 f. Dissertação (Mestrado em Design). UFPR, Curitiba, 2014. Disponível em: http://www.um.pro.br/prod/_pdf/001506.pdf. Acesso em: 8 maio 2021.

MATOS, A. Estudo sobre concepção e desenvolvimento de interfaces gráficas com a inserção de ux design. 2021. Disponível em: https://repositorio.pucgoias.edu.br/jspui/bitstream/123456789/1584/1/TCC%202%20_AlexandreRodrigues_final.pdf Acesso: 30 set 2021.

MORVILLE, Peter. User experience design. **Semantics Studios**, Scottsville, 21 de junho de 2004. Disponível em: http://semanticstudios.com/user_experience_design/. Acesso em: 20 jun. 2020.

PADOLSEY, James. **Clean Code in JavaScript**. Packt Publishing. 2020.

PADOVANI, Stephania; SCHLEMMER, André; SCARIOT, Cristiele A. Usabilidade & user experience. Uma discussão teórico-metodológica sobre comunalidades e diferenças. *In:* **Congresso Internacional de Ergonomia e Usabilidade de**

Interfaces Humano–Computador, p. 1-10, 2012. Disponível em: https://www.academia.edu/1869477/USABILIDADE_and_USER_EXPERIENCE_USABILIDADE_VERSUS_USER_EXPERIENCE_USABILIDADE_EM_USER_EXPERIENCE_UMA_DISCUSS%C3%83O_TE%C3%93RICO_METODOL%C3%93GICA_SOBRE_COMUNALIDADES_E_DIFEREN%C3%87AS. Acesso em: 9 jun. 2021.

Integrando *Clean Code* e técnicas de usabilidade para ampliar a qualidade

Autoria: Stella Marys Dornelas lamounier
Leitura crítica: Marco Ikuro Hisatomi

Objetivos

- Orientar os alunos sobre as técnicas de *Clean Code.*
- Facilitar a formatação de código a partir de boas práticas de *Clean Code.*
- Demonstrar testes automatizados de usabilidade.

1. A importância de um código bem escrito

Quando um código é bem escrito, consistente, pode-se observar que o programador se preocupou com os detalhes de forma a produzi-los com polidez, interpretável, eficiente e organizado. Deste modo, é notado que sua manutenção e manuseio podem ser realizados sem grandes dificuldades, ou seja, sem que o desenvolvedor passe horas tentando decifrar as funcionalidades que este código é capaz de realizar.

Técnicas de Clean Code devem ser aplicadas desde o início de sua concepção do código para que haja uma boa interpretação, a fim de possuir desde o início uma boa formatação para que sua comunicação seja de forma clara e objetiva.

É verdade que muitos códigos sofrem com degradação ao longo do tempo. Diversas pessoas podem alterar ou utilizar código por várias vezes, por isso é importante sempre deixá-lo limpo; desse modo, não sofreria tantos desgastes. Portanto, regras simples como melhorar nomes de variáveis, criar funções menores, eliminar repetição, reduzir if, são algumas das premissas que podem melhorar o código ao longo do tempo.

O desenvolvedor precisa atentar que os arquivos não devem ser longos, e tentar ao máximo trabalhar com arquivos menores, pois seu entendimento é mais fácil. Sendo assim, ao tratar da separação de estrutura, uma boa prática é separá-los com uma linha em branco, destacando cada grupo separado.

Nesse contexto, é de suma importância formatar o código, ou seja, separar a sua estrutura, de forma que se tenham códigos mais organizáveis possíveis. Deste modo, essa etapa não deve ser ignorada, pois sua responsabilidade é tratada como um elo de comunicação entre as regras de negócio e o desenvolvedor. Deve-se, ainda, levar em

consideração que criar um arquivo curto é muito mais entendível que um longo.

É importante o desenvolvedor atentar a algumas boas práticas em relação à indentação ou formatação, tanto vertical quanto horizontal, pois pequenos detalhes expostos nestes dois contextos podem trazer melhorias significativas para o código. Ou, quando não aplicados, pode transformar o projeto em uma catástrofe para usuários, ou seja, formatar deve sempre seguir as mesmas premissas estabelecidas pelo *Clean Code*, com regras simples, claras e consistentes.

Segundo De Jesus (2021), para um maior entendimento das estruturas e grupos, o ideal é criar uma separação de uma linha a cada nova estrutura obtendo-se, assim, um documento claro e de fácil entendimento após aplicação de indentação. Por exemplo:

```
public class GerarListaCompras {

   public static void main(String[] args) {

      int contador = 0;

      int[] listadeprodutos = new int [10];

      for (int numero : listadeprodutos)

         System.out.println(“Lista de produtos comprados: “ + numero);

      }}
```

O código acima, apesar de pequeno, não possui nenhum critério de indetação, o que pode dificultar o entendimento do bloco de código. Agora, vejamos o mesmo código utilização endentação.

```
public class GerarListaCompras {

   public static void main(String[] args) {

     int contador = 0;

     int[] listadeprodutos = new int[10];

     for (int numero : listadeprodutos) {

      System.out.println(“Lista de produtos comprados: “ + numero);

     }

   }

}
```

É perceptível que ao aplicar técnicas de indentação, o código, além de mais entendível fica mais leve e claro, facilitando, assim, as atividades dos desenvolvedores.

Mas, como é feita a leitura inicial de um código-fonte? De acordo com Martin (2019), sua leitura se parece muito com a leitura de um jornal de notícias, em que o leitor, primeiramente, preocupa-se em ler o topo de página do jornal, onde estão listados os assuntos mais importantes e principais, logo após uma sinapse que contenham dados sobre o contexto geral do documento. Posteriormente, ao passo que a leitura vai se afunilando, outros dados vão surgindo como data, autores, algumas citações, entre outros.

O mesmo pode ser percebido na leitura de um código. No topo, deve ser a ideia dos conceitos de alto nível e dos algoritmos sempre utilizando nomes simples e explicativos, com o aumento do detalhe, que devem

estar na parte de baixo do código contendo informações sobre as funções de mais baixo nível.

1.1 Formatação vertical

De acordo Martin (2019) é importante o programador, ao criar um sistema, ficar atento a pequenos detalhes de indentação, espaçamento, formatação vertical e horizontal. O código deve estar bem formatado e indentado, com regras simples e consistentes, com linhas em brancos para separar conceitos, como a separação de declarações importações de pacotes. Então, que tamanho deve ter um código-fonte?

Ainda segundo o autor, existe uma grande variedade de tamanhos de código, a depender do tipo de linguagem e projeto. Mas, tendem a ser no máximo 500 linhas e no mínimo 6; o ideal são blocos de aproximadamente 200 linhas.

Portanto, cada programador olhará o código de perspectivas diferentes, é subjetivo. Alguns conceitos são práticas de mercado, enquanto outros são baseados em vivência, experiência. O que deve ser levado em consideração é utilizar-se de boas práticas de *Clean Code* na programação para deixar um código aceitável e de fácil compreensão, para quem vai lê-lo ou utilizá-lo. E, ao sofrer alterações, que ele continue no mesmo patamar de qualidade deixado inicialmente.

Os espaçamentos são os principais responsáveis por separar conceitos, mas as linhas que possuem relacionamentos umas com as outras devem permanecer unidas. Não se deve separar por linhas em brancos ou por comentários duas ou três variáveis que faz parte do mesmo conjunto de blocos.

A declaração da variável, em técnicas de *Clean Code*, deve ser feita sempre mais próxima de onde vai ser utilizada; quando se tratar de variáveis locais, deve estar localizada sempre no topo das funções.

Por fim, quando estas variáveis são de controle de loops é importante declará-las dentro da própria estrutura de interação conforme o exemplo a seguir:

```
public int contadorCasoTeste () {

    int contador = 0;

    for ( Cada Teste : testes)

      contador += each.contadorCasoTeste();

    return contador;

}
```

Observe que onde está localizada a variável contador, dentro do bloco que ela realizará a ação de contagem. Quando se tratar de funções, se elas forem dependentes, devem seguir a mesma premissa das variáveis acima, isto é, deve estar sempre próxima umas das outras, onde a função a que chamar deve sempre estar sobreposta à função que é chamada.

1.2 Formatação horizontal

Linhas de código devem ter tamanhos? Basicamente sim. O desenvolver deve estar focado em manter linhas curtas que fiquem entre 100 ou até 120 caracteres.

A formatação horizontal também utiliza o espaçamento em branco para realizar a separação do código, pois separar elementos pode facilitar a leitura e entendimento do código. Assim:

```
int TamanhoNome = nome.length ( ) ;

totalCaracteres += TamanhoNome;
```

Note a importância da separação dos atributos por espaços em branco; sua responsabilidade é separar os elementos da direita de elementos da esquerda a fim de proporcionar facilidade na leitura.

Outra observação, segundo Martin (2019), é atentar a questões como: não se deve separar com espaços em brancos nomes de funções e os parênteses que elas possuem, isso devido ao elo entre as partes.

Quando se tratar de alinhamento horizontal com muitas variáveis é recomendado que elas devem ser seguidas umas da outras sempre no mesmo nível de endentação, conforme demonstrado a seguir:

```
Public Class Document implements Juristic {

    private Integer    id;

    private TypeDocument typeDocument;

    private Theme themeDocumen;

    private Request          request;

}
```

É observado que o bloco possui várias variáveis que possuem nomes extensos para leitura, o que pode dificultar a visualização de valores, criando obstáculos para o entendimento. Por isso é importante o alinhamento em que as atribuições e declarações devem pertencer a um mesmo nível de indentação.

1.3 Tratamento de erros

Execuções não tratadas podem ocasionar situações inesperadas, utilizar-se de entrada de dado de forma errônea e, deste modo, sistemas computacionais podem falhar de uma hora para a outra. É normal dar errado, mas a responsabilidade é do desenvolvedor de certificar-se que o código faça o que deve ser feito quando isso acontecer.

Algumas linguagens de programação no passado não suportavam as exceções, e havia poucas técnicas para informar estes erros. Eram utilizadas para alterar o fluxo dos programas ou evitar falhas em equipamentos. Por exemplo, o que aconteceu com o foguete Ariane 5 (1996) foi uma falha de exceção overflow não tratada e outros bugs que marcaram a história dos softwares, provavelmente ou foram exceções não tratadas ou tratadas de forma incorreta. Deste modo é relevante saber o conceito de exceção:

Uma exceção representa um comportamento anormal, indesejado, que ocorre raramente e requer alguma ação imediata, um tratamento adequado, em uma parte do programa. Uma exceção indica uma condição de erro, tal como um overflow aritmético, uma divisão por zero, uma referência a um objeto nulo. Uma exceção também pode indicar uma ocorrência anormal, mas que não é considerada um erro, tal como o fim do arquivo. (SIQUEIRA, 2014, [s.p.])

Entretanto, engana-se que estes erros ocorrem apenas nas condições de softwares; podem acontecer em situações imprevisíveis que interferem na execução dos programas, como quedas de conexões de rede ou problemas de hardware.

Com a evolução das linguagens de programação, em especial com a ascensão da programação orientada a objetos, as exceções passaram a ser criadas quando o erro fosse encontrado, criando, assim, códigos mais entendíveis e claros. Por isso é importante criar exceções em vez

de código que retornem erros, e o Clean Code fornece algumas boas práticas para deixar o tratamento de exceções ainda mais fácil de manipular.

Exceções podem e devem ser tratadas de forma que sejam capazes de permitir ou não a continuação de procedimentos, unidades, de uma sub-rotina ou função que a invocou. Por isso é importante estabelecer o conceito de tratamento de exceção:

> O tratamento de exceções é uma boa prática que deve ser perseguida pelos programadores. Todo programa deve ser implementado de forma que todas as exceções possíveis de ocorrer sejam tratadas adequadamente. Esta prática aumenta a confiabilidade do programa na medida em que o programa é capaz de tratar os possíveis erros que possam ocorrer durante sua execução sem perda de informações ou comprometer as transações de negócio da empresa. Infelizmente, muitos programadores não dão a devida atenção a esta boa prática e o que temos como resultado são programas e às vezes até mesmo aplicações inteiras que são frágeis e vulneráveis do ponto de vista de tratamento de erros. (SIQUEIRA, 2014, [s.p.])

A seguir, apresentaremos algumas práticas que podem auxiliar os desenvolvedores a fazer o uso das exceções para manter o código limpo e de fácil manutenção. Geralmente, esses tratamentos não devem jamais atrapalhar a lógica já existente, e sim possuir mensagens capazes de facilitar a organização e localização do problema, auxiliando, assim, na composição de testes de software.

Martin (2019) relata que uma boa prática para manter os tratamentos de erros limpos e organizados é não lançar um emaranhado de códigos de erros nas funções ou métodos, como era feito pelas linguagens de programação mais antigas, e sim disparar de forma organizada estas exceções.

Primeiramente, deve-se criar a estrutura de *try-catch-finally*, onde o *try* é responsável pelas transações que podem ser canceladas a qualquer

momento. Quando isso acontecer, é retornado o *catch*. Neste contexto, quando uma exceção for lançada, ela deve ser capaz de determinar corretamente onde o erro está localizado. O *catch* é responsável pela definição de escopo na execução, dando mais clareza ao código pelo fato de sempre apresentar resultados. O trecho de código criado por Neves (2019) demonstra a má utilização do Try-catch:

```
try {

  FuncaoDeExcecao();

} catch (error) {

  console.log(error);

}
```

O ideal é a utilização de trechos que, após a definição do fluxo normal, e caso haja a necessidade da criação de fluxos alternativos, estes devem conter exceções não só com apenas comandos de System.out.println() ou um console.log(), conforme demonstrado por Neves (2019):

```
try {

  FuncaoDeExcecao();

} catch (error) {

  // Uma opção (mais chamativa que console.log):

  console.error(error);

  // Outra opção:

  NotificarUsuario(error);
```

```
  // Outra opção:

  ReportarParaOServico(error);

  // OU as três!

}
```

Lembre-se o usuário do código deve entender o que está escrito mesmo tratando-se de erros e exceções. Neste contexto, deve-se disparar alertas para que o responsável seja capaz de identificar os erros e tentar amenizá-los de forma rápida.

Os erros podem vir de diferentes fontes e, assim, possuir diversos mecanismos de classificação, por exemplo: erros de codificação; falhas em dispositivos de hardware ou na rede. A preocupação deve ser criar classes de exceção e como elas irão capturar estes erros.

Outra boa prática é não passar nem retornar NULL. Imagine um código fonte repleto de validações nulls – jamais faça isso –, em que basta disparar e esquecer como foi feita a verificação que tudo estará perdido. Apenas faça isso se uma API espera que você faça isso; do contrário, jamais o utilize ao fazer isso, conforme pode ser analisado no trecho de código a seguir:

```
public void registrarCompra(Compra compra{

  if (compra != null) {

    ItemCompra registro = persistentStore

     .getItemCompra();

    if (registro!= null) {

      Item registro = registro
```

```
    .getItem(compra.getID());
  if (existe.getBillingPeriod()
  .hasRetailOwner()) {
    existe.registro(compra);
    }
   }
  }
 }
```

É observado que o trecho de código não possui erros, mas temos o null implementado mais de uma vez, e ao ser executado, correríamos o risco de não entendermos porque foi criado e quando isso acontecer. Assim, pode-se ter como resultado um NullPointerException que trata de uma exceção lançada quando um programa tenta acessar um objeto de memória que não foi instanciado, ou seja, não possui valor definido.

1.4 Separação e construção de sistemas

Martin (2019), relata que existem quatro regras simples que auxiliam os desenvolvedores na criação de bons projetos utilizando *Clean Code* que devem ser sempre a premissa de códigos limpos, entendíveis e simples:

1. Efetuar todos os testes.
2. Sem duplicação de código.
3. Expressar o propósito do programador.
4. Minimizar o número de classes e métodos.

Regra 1: qualquer sistema computacional deve possuir qualidade o suficiente para atender todas as exigências do cliente, e claro, deve funcionar conforme os requisitos levantados. Neste contexto, o software deve passar por todos os testes, sejam eles manuais ou automatizados, sistemas computacionais adequados devem ser passíveis de testes, contemplando os requisitos funcionais e não funcionais.

Regras de 2 a 4: códigos e classes devem ser limpos, e para que isso aconteça é importante refatorar sempre que necessário. Utilizar essa técnica traz ao desenvolvedor a liberdade de reduzir e minimizar classes e funções. Além disso, melhorar nomes e métodos, diminuir acoplamento e eliminar duplicação ineficazes que diminuem a qualidade do sistema, aumenta seu trabalho e sua complexidade. Temos, também, casos de linhas semelhantes que podem trazer ainda mais confusão para o usuário como mostrado no trecho de código descrito pelo autor.

```
public int Size()
   {
      // Size implementation
   }


   public bool IsEmpty()
   {
      // IsEmpty implementation
   }
}
```

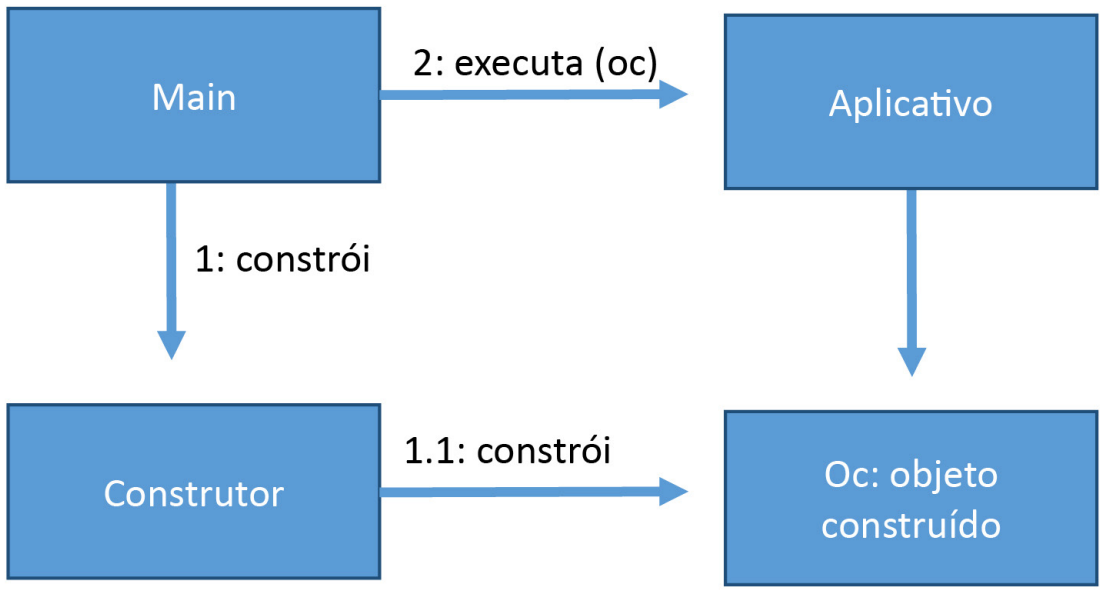
Agora, vejamos o mesmo código onde cada método é separado. Elimina-se a duplicação e aplicamos o isEmpty na declaração do size.

```
 public bool IsEmpty()
   {
      return Size() == 0;
   }
}
```

Outro ponto importante que o desenvolvedor deve observar é a separação da inicialização com o uso do sistema, ou seja, deve-se separar processos de inicialização, dos objetos e conexões.

A Figura 1 ilustra um diagrama de fluxo de controle, que inicia com a função Main responsável por construir os objetos para a construção do sistema (item 1). Posteriormente, à sua direita, passa-se ao aplicativo que o utiliza. Esse aplicativo e o objeto construído estão longe do Main, o que significa que o aplicativo ou todo seu processo de construção não enxerga, mas sim fica na expectativa que tudo deve ser devidamente construído.

Figura 1 – Separação de construção de objetos



Fonte: adaptada de Martin (2019, p. 156).

Outro modelo que pode ser utilizado é o Padrão Factories (*Abstract Factory*), proposto por GOF (1994) de Padrões de Projeto, em que também se tem todas as dependências apontadas pelo Main.

1.5 Princípio da responsabilidade única

Sabemos que milhares de códigos são escritos diariamente por diferentes tipos de desenvolvedores. Fazê-los funcionar já não é suficiente, deve-se possuir padrões de conceitos de Clean Code com boas práticas de programação de suas linhas, a fim de torná-las legíveis e organizadas. Dessa forma, alterações são comuns, e podem ocorrer por vários anos ao longo do tempo de vida de um sistema computacional. Uma destas técnicas utilizadas por desenvolvedores de sistemas que utilizam Clean Code é o princípio da responsabilidade única, ou SRP, que pode ser definido como:

> 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”. (MAGALHÃES; TIOSSO, 2019, p. 7)

Neste contexto, deve-se observar que classes podem ser modificadas sempre que necessário, e isso acontece frequentemente aumentando seu tamanho com a criação de novos métodos. É importante analisar e refatorar para não confundir o código e utilizar-se da SRP como aliada na construção de classes mais organizadas.

O princípio da responsabilidade única é auxiliar na criação de softwares melhores e mais organizados, em que classes que contenham apenas uma responsabilidade, terão muito menos casos de testes, levando, assim, menos tempo em testes de software.

Outras características com menos funcionalidades esta classe terá muito menor independência e menos acoplamento. Assim, serão muito mais organizadas, pois são menores e mais fáceis de realizar leitura e pesquisa, conforme podemos observar no bloco de código a seguir:

```
public class Book {

    public string Nome {get;set; }

    public string Autor {get;set; }

    public string Texto {get;set; }

public string ReplaceWordInText (string word) {

        return Text.Replace (word, Text);

    }

public bool IsWordInText (string word) {

        return Text.Contains (word);

 }
```

Um trecho de código em que a classe denominada Book, tem como responsabilidade inicial armazenar dados de um livro quantas vezes forem necessárias. Entretanto, ao mesmo tempo, ela também possui métodos para a impressão dos dados de livro, violando, assim, o SRP. O ideal é que seja criada uma classe separada que tenha como responsabilidade a impressão dos dados do livro conforme demonstrado no bloco de código a seguir.

```
public class BookPrinter {

    public void PrintTextToConsole (string text)
```

```
    {
        Console.WriteLine ($" Nome do livro: {text}")
    }
      public void PrintTextToAnotherMedium(string text) (string text)
      {
        Console.WriteLine ($" {text}")
    }
}
```

O código demonstra a criação de uma classe chamada BookPrinter que tem como responsabilidade imprimir os dados do livro armazenado. Ao utilizar desta separação, nota-se que cada uma das classes possui sua própria responsabilidade.

Por fim, muitas são as boas práticas que devem ser seguidas pelos profissionais que desejam criar códigos limpos de fácil interpretação, com qualidade tanto para o usuário final quanto para o próprio desenvolvedor. Aplicar técnicas de *Clean Code* ao longo do código é sinal positivo para construção de sistemas que carreguem qualidade e organização.

2. Ferramentas de usabilidade

Sabemos que a usabilidade em sistemas computacionais tem como objetivo aliar facilidade de uso e satisfação com a utilização do software, mas, ao mesmo tempo é notado que medir a usabilidade tem se tornado complexo e muitas vezes subjetivo. Desse modo, é importante

o emprego e a utilização de algumas ferramentas que podem auxiliar os profissionais da área a medir o quão o usuário está satisfeito com o que lhe foi entregue.

Existem algumas ferramentas no mercado que podem auxiliar ainda mais a testar a usabilidade em sistemas computacionais, tais como:

Loop 11: ferramenta de usabilidade que permite executar os testes em sites ou protótipos para garantir que os usuários tenham boa experiência com o Sistema. A partir da aplicação com usuários reais é possível analisar as tarefas e rever o site. Está disponível de forma gratuita no portal Loop 11.

Fivesecondtest: realiza testes de usabilidade para detectar ambiguidades ou problemas com sua interface, além de ser possível realizar interações com usuários e compartilhar opiniões, criando relatórios e avaliações. A ferramenta pode ser utilizada de forma on-line disponível no seu portal Fivesecondtest.

Google Analytics: também pode realizar testes de usabilidade de forma gratuita, analisando, por meio de insights, o comportamento na navegação de *websites* gerando relatórios que auxiliam na avaliação do site. Está disponível na plataforma Google.

Enfim, inúmeras são as ferramentas que auxiliam as empresas de desenvolvimento a criarem software cada vez mais fáceis de utilização e com qualidade. O que cada uma deve fazer é escolher o modelo e a metodologia que mais se adaptar a sua realidade sem deixar de pensar no usuário final.

Referências

DE JESUS, L. S. Clean Code (código limpo). **Teste a Velocidade**, [s.l.], 15 de fevereiro de 2021. Disponível em: https://testeavelocidade.com.br/codigo-limpo/. Acesso em: 2 jul. 2021.

FIVESECONDTEST. **Five second tests**. Disponível em: https://fivesecondtest.com/. Acesso em: 22 jul. 2021.

GOOGLE ANALYTICS. **Analytics**. Disponível em: https://analytics.google.com/. Acesso em: 22 jul. 2021.

LOOP11. **Real people. Real tasks. Unreal websites**!. Disponível em: https://www.loop11.com/. Acesso em: 22 jul. 2021.

MAGALHÃES, P. A.; TIOSSO, F. Código Limpo: padrões e técnicas no desenvolvimento de software. **Revista Interface Tecnológica**, Taquaritinga, v. 16, n. 1, p. 197-207, 2019. Disponível em: https://revista.fatectq.edu.br/index.php/interfacetecnologica/article/view/597. Acesso em: 22 jul. 2021.

NEVES, Y. P. **Clean Code–Tratamento de erros**. 2019. Disponível em: http://tech.azi.com.br/clean-code-tratamento-de-erros/. Acesso em: 3 jul. 2021.

SIQUEIRA, J. F. R. **Tratamento de Erros e Exceções**. 2014. Disponível em: https://sites.google.com/site/proffernandodesiqueira/disciplinas/paradigmas-de-linguagens-de-programacao/aula-6—-tratamento-de-erros-e-excecoes. Acesso em: 3 jul. 2021.

BONS ESTUDOS!