RAFAELA SOARES NUNES

A CONTRIBUIÇÃO DOS PRINCÍPIOS DE PROGRAMAÇÃO NO DESENVOLVIMENTO DE SOFTWARE SUSTENTÁVEL

Monografia apresentada ao Programa de Educação Continuada da Escola Politécnica da Universidade de São Paulo, para obtenção do título de Especialista, pelo Programa de MBA USP Tecnologias Digitais e Inovação Sustentável.

SÃO PAULO

2020

RAFAELA SOARES NUNES

A CONTRIBUIÇÃO DOS PRINCÍPIOS DE PROGRAMAÇÃO NO DESENVOLVIMENTO DE SOFTWARE SUSTENTÁVEL

Monografia apresentada ao Programa de Educação Continuada da Escola Politécnica da Universidade de São Paulo, para obtenção do título de Especialista, pelo Programa de MBA USP Tecnologias Digitais e Inovação Sustentável.

Orientadora: Ma. Márcia Cristina Machado

SÃO PAULO

2020

Catalogação na publicação

Nunes, Rafaela Soares
A contribuição dos princípios de programação no desenvolvimento de software sustentável, Rafael Soares Nunes; orientadora, Márcia Cristina Machado. – 2020

75f.

Monografia (MBA em Tecnologias Digitais e Inovação Sustentável) - Escola Politécnica da Universidade de São Paulo. PECE – Programa de Educação Continuada em Engenharia.

1.Ciclo de Vida, 2. Desenvolvimento de Software Sustentável, 3. Padrão de Desenvolvimento, 4. Princípios do SOLID, 5. Software, 6. Sustentabilidade. I - Universidade de São Paulo. Escola Politécnica. PECE – Programa de Educação Continuada em Engenharia II.t.

AGRADECIMENTOS

Por meio desta, expresso meus sinceros agradecimentos à minha orientadora professora Márcia Cristina Machado por todo o seu apoio, orientação, contribuição e paciência para que este trabalho acontecesse. A sua experiência e postura profissional me inspiraram e me possibilitaram ampliar meus horizontes e superar meus desafios para atingir o objetivo com o presente estudo.

Agradeço imensamente ao meu companheiro de vida, Leandro Frata dos Santos, por todo seu apoio, paciência e compreensão nos momentos mais difíceis ou conturbados em que precisei de apoio ou simplesmente alguém para conservar, me ouvir, ele sempre está ao meu lado seja em momentos bons, seja em momentos ruins. Com isso, sei que nunca estou sozinha assim como ele também nunca está sozinho, porque sempre temos um ao outro.

E por fim, mas não menos importantes meus profundos agradecimentos ao meu pai Manuel Brito Nunes e a minha mãe Delvaides Lira Soares Nunes, por sempre acreditarem em mim, pelo apoio incondicional e pelo o exemplo de força, dedicação, disciplina e persistência que sempre me passaram e fizeram com que me tornasse a pessoa que sou hoje, capaz de enfrentar desafios com a paciência e persistência necessárias para alcançar meus sonhos. Agradeço a todos os esforços dos meus pais para que eu tivesse a oportunidade de estudar e obter um título de formação acadêmica, que eles mesmos não tiveram a oportunidade de adquirir.

RESUMO

O desenvolvimento de *software* é pouco associado aos impactos ambientais e econômicos da sustentabilidade. Mas a maneira como um sistema é construído tem grande importância sobre como os recursos computacionais, energéticos e financeiros que suportam o seu ecossistema serão utilizados. Isto posto, um software pode ser sustentável se desenvolvido de forma orientada a minimizar ou eliminar os impactos negativos ao meio ambiente e as pessoas. Diante desta perspectiva, o presente trabalho tem por objetivo comparar os resultados obtidos com o uso de princípios e padrões da programação atuais com os resultados esperados de um *software* sustentável. A finalidade é identificar as boas práticas de mercado que contribuem para o desenvolvimento de *software* sustentável, para motivar ou desmotivar a sua adoção, do ponto de vista da sustentabilidade. Por meio da pesquisa bibliográfica sobre o ciclo de vida do desenvolvimento de *software*, eficiência energética, sustentabilidade e padrões de desenvolvimento de *software*, buscou-se fazer uma análise comparativa entre os resultados do uso de padrões de desenvolvimento adotados nos modelos de programação orientada a objetos e os resultados esperados de um *software* sustentável. Como resultado destas análises, neste estudo conclui-se que o uso dos padrões de desenvolvimento existentes é adequado para atingir o objetivo de produzir *softwares* sustentáveis, sem a necessidade da elaboração de um padrão específico de desenvolvimento para esta finalidade.

Palavras-chave: Ciclo de Vida, Desenvolvimento de *Software* Sustentável, Padrão de desenvolvimento, Princípios do SOLID, *Software*, Sustentabilidade.

ABSTRACT

Software development is little associated with the environmental and economic impacts of sustainability. But the way a system is built has great importance on how the computational, energy and financial resources that support its ecosystem will be used. That said, software can be sustainable if developed in a way aimed at minimizing or eliminating negative impacts on the environment and people. Given this perspective, the present work aims to compare the results obtained with the use of current programming principles and standards with the expected results of sustainable software. The indicated one is to identify as good market practices that contribute to the sustainable software development and to motivate or discourage its adoption, from the point of view of sustainability. Through bibliographic research on the software development life cycle, efficiency, sustainability, and software development standards, we sought to make a comparative analysis between the results of using development patterns of object-oriented languages and the expected results of sustainable software. As a result of these analyzes, this study concludes that the use of existing development standards is adequate to achieve the objective of producing sustainable software, without the need to develop a specific development standard for this purpose.

Keywords: Development standard, Life Cycle, Software, Solid Principles, Sustainability, Sustainable Software Development.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CMMAD	Comissão Mundial sobre Meio Ambiente e Desenvolvimento
DDD	Domain-Driven Design
DIP	Dependency Inversion Principle
HTTP	HyperText Transfer Protocol
IEC	International Electrotechnical Commission
IEEE	Instituto de Engenheiros Eletricistas e Eletrônicos
ISSO	International Organization for Standardization
ISP	Interface Segregation Principle
LSP	Liskov Substitution Principle
MVC	Model-View-Controller
NBR	Norma Técnica Brasileira
OCP	Open-Closed Principle
POO	Programação Orientada a Objetos
REST	Representation State Transfer
SRP	Single Responsability Principle
TI	Tecnologia da Informação
XP	Extreme Programing

SUMÁRIO

1 INTRODUÇÃO

O *software* sustentável segundo Dick, Naumann e Kuhn, 2010 é um *software* cujos impactos negativos à economia, à sociedade e ao meio ambiente durante o seu desenvolvimento, implantação e utilização são mínimos ou causam efeitos positivos em relação à sustentabilidade. A sustentabilidade por sua vez é um termo cada vez mais presente, seja nas grandes cadeias produtivas ou em pequenas ações do dia-a-dia. De acordo com Veiga (2019) o termo sustentabilidade é usado para exprimir uma vaga ambição futura de continuidade, durabilidade ou perenidade. Segundo pesquisas realizadas por Nidumolu, Ram e Prahalad (2014) a sustentabilidade mostra-se como uma nova frente de inovação para as companhias, que os autores denominaram como "empresas inteligentes", por trazer retornos financeiros e até criar novos negócios.

A popularização da mentalidade sustentável atinge diversos setores na área TI (Tecnologia da Informação) e foi impulsionada com a TI Verde, que representa as iniciativas sustentáveis neste setor. Como enfatizado por Calero e Piattini (2015) a TI Verde está se tornando uma necessidade, à medida que mais e mais organizações estão implementando algum mecanismo de soluções sustentáveis, e dentro dessas iniciativas está o desenvolvimento de *software* sustentável, que embora não gere um produto final tangível, gera impactos com a utilização de recursos computacionais e energéticos.

Neste trabalho aborda-se a sustentabilidade aplicada no desenvolvimento de *software*, sobre o qual se apresenta uma análise comparativa dos princípios de padrões de programação atuais com os conceitos do desenvolvimento de *software* sustentável, buscando-se identificar quais os princípios de programação orientada a objetos são promotores do *software* sustentável.

1.1 Motivação

Segundo Nidumolu, Ram e Prahalad (2014) a sustentabilidade é uma questão que já transformou o cenário competitivo, e tende a forçar as organizações a reformular a maneira como criam/elaboram seus produtos, tecnologias, processos e modelo de negócios. Partindo desta ideia, incluí-la nas atividades ordinárias que realizamos pode trazer não somente contribuições ao indivíduo, mas à organização e/ou a sociedade.

Na atualidade o *software* está presente em diversos momentos do cotidiano de grande parte da população mundial. Em sua obra Calero e Piattini (2015) citam a importância da conscientização de todos os envolvidos no desenvolvimento, na compra e na utilização de um *software* sustentável. Perante ao crescimento do número de *softwares* que acompanham a evolução tecnológica, surge a questão que orienta a pesquisa deste estudo: **Quais os princípios de padrões da programação adotam práticas sustentáveis e como contribuem para desenvolvimento de *softwares* sustentáveis**?

1.2 Objetivo

O objetivo deste trabalho é identificar quais os princípios ou padrões de programação são promotores da criação de *softwares* sustentáveis. Para tanto, será realizada a comparação dos resultados obtidos na utilização destes princípios com os objetivos esperados de um *software* sustentável.

Este estudo pretende ainda conscientizar os profissionais envolvidos na construção de *softwares*, promover a utilização de boas práticas já existentes na programação, auferindo resultados positivos à sustentabilidade que estes podem trazer com a sua devida utilização.

1.2.1 Objetivos específicos

Os objetivos específicos deste estudo estão listados abaixo.

- identificar os princípios utilizados no desenvolvimento de *software dentro do paradigma da orientação à objetos*;
- identificar os conceitos utilizados no desenvolvimento de *software* sustentável;
- comparar os resultados e identificar quais são os que contribuem para o desenvolvimento de *software* sustentável.

1.3 Justificativa

Segundo Calero e Piattini (2015) a eficiência energética nunca foi um requisito essencial no desenvolvimento de *software*, por esse motivo possui um enorme potencial a ser explorado. Requisito este que está diretamente ligado aos impactos ambientais do desenvolvimento, e geralmente são considerados requisitos não funcionais de sistemas, do mesmo modo que os requisitos de segurança também o são. (BETZ; CAPORALE, 2015).

Existem muitos estudos sobre o *software* sustentável escritos em língua inglesa e poucos na língua portuguesa, conforme observado em pesquisa realizada nas bases de dados Scopus e IEEE em abril de 2020. Conforme Calero e Piattini (2015) no processo de desenvolvimento de *software* raramente a sustentabilidade é uma temática relevante. Logo, este trabalho almeja contribuir para esta área de conhecimento, com a tradução dos conceitos do *software* sustentável em princípios e paradigmas da programação existentes, além de amplamente utilizados no mercado profissional.

Outra questão relevante é que a adoção de princípios ou padrões de programação nem sempre é obrigatória nas empresas, cabendo ao programador a decisão de usar ou não estes modelos. Sendo assim, conforme preconizado por Calero e Piattini (2015) é dever dos profissionais da área conscientizar e obter modelos, métodos e ferramentas que reduzam o impacto ambiental dos processos de desenvolvimento de *software*. Deste modo, ao relacionar os princípios de padrões com a sustentabilidade é desejado aumentar a relevância dos mesmos na tomada de decisão da sua utilização.

1.4 Contribuição

Neste trabalho é esperada a contribuição nas áreas de conhecimento do desenvolvimento de *software* e da sustentabilidade, por meio da conscientização e popularização do tema desenvolvimento de *software* sustentável.

Ao identificar quais os princípios ou padrões da programação são favoráveis para se atingir um *software* sustentável, espera-se traduzir os conceitos do *software* sustentável em princípios já conhecidos pelos profissionais da área. E dessa forma, simplificar a compreensão e fomentar a adoção do tema.

Ademais por meio de uma análise comparativa entre os resultados alcançáveis com o uso de padrões de desenvolvimento e os resultados esperados em um *software* sustentável, espera-se verificar se o uso dessas boas práticas elimina a necessidade da criação de um framework específico para conduzir ao desenvolvimento de *softwares* sustentáveis.

1.5 Metodologia

O método de pesquisa aplicado no estudo é exploratório. Foi realizada uma revisão bibliográfica da literatura, organizada nas áreas de conhecimento da engenharia de *software* e da sustentabilidade. As sub-áreas citadas são *software* sustentável, ciclo de vida do *software*, padrões e princípios de desenvolvimento de *software*.

A pesquisa bibliográfica considerou o estado atual da arte relacionado aos temas supracitados, além de explorar os seus princípios e conceitos, e os resultados esperados com a suas adoções (SÁ-SILVA; ALMEIDA; GUINDANI, 2009; WAZLAWICK, 2009).

A pesquisa também contou com uma survey realizada com a participação de profissionais da área de desenvolvimento de *software*, tendo como finalidade a identificação do nível de familiariadade dos profissionais com o uso de padrões, tecnologias e conceitos de sustentabilidade (GREEN; KENNEDY; MCGOWN, 2002; SÁ-SILVA; ALMEIDA; GUINDANI, 2009).

1.6 Organização do trabalho

O trabalho está dividido em quatro capítulos, iniciando pela introdução, seguindo para o referencial teórico e desenvolvimento da pesquisa, e fechando com a conclusão. Na introdução é apresentado o contexto do trabalho, motivações, justificativas, seu objetivos e metodologias utilizadas.

No segundo capítulo o referencial teórico traz todo o embasamento científico e definições dos conceitos chave do trabalho, e trabalhos correlatos. O terceiro capítulo apresenta o desenvolvimento em si, onde são descritas as pesquisas, as análises comparativas realizadas e seus respectivos resultados.

Por fim, no quarto capítulo mostra-se a conclusão do estudo, e as contribuições resultantes, bem como as sugestões para trabalhos futuros.

2 REFERENCIAL TEÓRICO

Neste capítulo apresenta-se os conceitos sobre a sustentabilidade, a definição de *software* sustentável, as normativas envolvidas no ciclo de vida do *software*, a definição do termo TI Verde, padrões, princípios e boas práticas de programação, assim como trabalhos correlatos.

Os temas foram ordenados pela dependência conceitual, começando pela área de conhecimento da sustentabilidade, a sua intersecção com a área de engenharia de *software* e por fim os princípios e paradigmas da engenharia de *software* que podem contribuir com as características sustentáveis de uma aplicação.

2.1 Sustentabilidade

A sustentabilidade abrange três aspectos ou dimensões, o econômico, o social e o ambiental, sendo este último o que mais contribuiu para a divulgação do termo. No ambiente corporativo a junção das três dimensões da sustentabilidade foi chamada por Elkington (2011) de "*triple bottom line*". A presente monografia aborda dois aspectos da sustentabilidade, o econômico e o ambiental nos quais o desenvolvimento de *software* tende atingir maior relevância, pela forte ligação com a eficiência energética no consumo do processamento de um sistema, conforme sugerido por Calero e Piattini (2015).

A sustentabilidade de acordo com Gonçalves-Dias (2014) "significa aquilo que pode ou deve se sustentar. Além disso, qualifica a capacidade de se manter constante ou estável por longo período". Entretanto de acordo com Veiga (2019), o termo sustentabilidade que surgiu com a questão ambiental da sobrepesca, foi banalizado na atualidade e é difícil dizer que tenha apenas uma única definição. Já o termo desenvolvimento sustentável surgiu em 1987 na Comissão Mundial sobre Meio Ambiente e Desenvolvimento (CMMAD) e foi definido pela

comissão lidera por Brundtland (1987, p,16) como "...que satisfaz as necessidades do presente sem comprometer a capacidade das gerações futuras de satisfazer suas próprias necessidades"[1].

2.2 Sustentabilidade e Tecnologia (TI Verde)

A TI (Tecnologia da Informação) Verde ou do inglês “Green IT” é um termo usado para associar sustentabilidade às atividades e produtos gerados pela área de TI. Segundo Murugesan e Gangadharan (2012) “TI Verde é alcançar a viabilidade econômica e melhorar o desempenho e o uso do sistema, respeitando nossas responsabilidades sociais e éticas”. A TI Verde pode estar presente em várias etapas da TI, como por exemplo no descarte correto de equipamentos ou na utilização eficiente de recursos energéticos ao manter os dispositivos ligados somente quando necessário.

Com o aumento da digitalização nas organizações a TI tornou-se ser uma forte aliada na sustentabilidade dos negócios, viabilizando o trabalho remoto e as vídeo conferências, que contribuem para redução de muitos deslocamentos, implicando na redução de emissões de gases do efeito estufa (MOLLA; COOPER, 2009).

Outro fator que pode ser explorado na TI Verde é a ecoeficiência ou uso eficiente de recursos, com otimização do uso de recursos energéticos tanto na própria indústria de TI (*data centers*), quanto em outras indústrias, adotando-se por exemplo a automatização com sistemas inteligentes de economia de energia (JONES et al., 2013).

Além de abordar o consumo de energia nos *data centers* e em equipamentos de TI deve-se considerar o uso da água que também é muito relevante, visto que o sistema de refrigeração e resfriamento necessários para o adequado funcionamento dos equipamentos consome um considerável volume de água. No caso do Brasil observa-se que a economia de recursos energéticos pode viabilizar a economia de água, já que a maior parte da matriz energética do país vem de hidrelétricas. Ademais o uso de componentes químicos na produção de equipamentos eletrônicos tem alto impacto ambiental se descartado de forma inadequada,

[1] Humanity has the ability to make development sustainable to ensure that it meets the needs of the present without compromising the ability of future generations to meet their own needs

gerando a contaminação do solo e dos recursos híbridos (JONES et al., 2013; LAMAS; GIACAGLIA, 2013; SHARMA et al., 2009).

A prática da TI verde nas fábricas de produtos eletrônicos ocorre quando nos processos produtivos estas empresas optam por não utilizar ou reduzir o uso de substâncias nocivas ao meio ambiente, ao reduzir o consumo ou buscar por alternativas às matérias-primas finitas, ou ainda quando fazem o reuso de metais preciosos. Outro fator contribuinte está no desenvolvimento de produtos com projetos ecologicamente corretos, e na adoção de design que favoreça a reciclagem dos equipamentos. Já as demais organizações podem contribuir promovendo o consumo consciente dos equipamentos eletrônicos, comprando de fabricantes que utilizam menos substâncias nocivas e fazendo o descarte correto e a reutilização destes equipamentos (JONES et al., 2013; MURUGESAN; GANGADHARAN, 2012).

2.3 Software Sustentável

O conceito de *software* sustentável que o trabalho aborda é o *software* desenvolvido de maneira orientada a sustentabilidade. Conforme Kern et al. (2018) a existência dos seguintes critérios pode ser considerada uma forma de identificar se um *software* é ou não sustentável: critérios de qualidade de *software*; a existência de critérios de sustentabilidade como eficiência energética; e adoção de critérios de sustentabilidade nas fases que compõem o ciclo de vida de desenvolvimento de *software*.

Com desempenho eficiente de processamento que implica economia de recursos energéticos, e agilidade na manutenção decorrente de uma construção focada na reusabilidade, um *software* pode ser considerado mais sustentável do que outro *software* com funcionalidade similar, porém sem atingir os mesmos requisitos de eficiência energética e reuso, de acordo com estudo realizado por Procaccianti, Fernández e Lago (2016).

Além das questões relacionadas ao uso eficiente dos recursos ambientais a fim de manter a perenidade destes recursos, ou seja, usá-los de forma que não falte às próximas gerações, o ciclo de desenvolvimento de *software* apresenta impactos econômicos com a redução do consumo energético de seu processamento. Ademais um *software* sustentável pode gerar impactos sociais tanto para as pessoas que o utilizam quanto para as pessoas que o desenvolvem (CALERO; BERTOA; MORAGA, 2013; CHANG; LEE; CHEN, 2014).

Quando um *software* for desenvolvido de maneira a facilitar a sua manutenção e possibilitar a reutilização de código, poderá impactar a motivação e produtividade do desenvolvedor, e consequentemente, viabilizar a redução de custos com desenvolvimento. Um *software* desenvolvido com foco na usabilidade e que possibilita o acesso à informação ou serviço por pessoas que antes não os tinham, pode ser entendido como gerador de grande impacto social (GARCÍA-RODRÍGUEZ DE GUZMÁN; PIATTINI; PÉREZ-CASTILLO, 2015; O'BRIEN; DOIG; CLIFT, 1996; SUWANYA; KURUTACH, 2008).

2.3.1 Características e Papéis no Ciclo de Vida do Software Sustentável

Cada profissional envolvido no desenvolvimento, teste, implantação, monitoração e manutenção de um *software* sustentável tem o dever de aprimorar e transmitir conhecimentos que possam melhorar continuamente os processos para mitigar ou eliminar impactos ambientais, assistir seus colegas nestas questões e fazer escolhas ecologicamente corretas quando possível na tomada de decisões (KATZ et al., 2019; MORAGA et al., 2017).

Tanto os usuários finais de um *software* quanto os envolvidos direta ou indiretamente na sua construção e manutenção, têm responsabilidade de resolver conflitos de interesse com impactos ambientais, a fim de diminuir ou suprimir esses impactos em detrimento de escolhas relacionadas a usabilidade, funcionalidade e aparência da aplicação, dentre outros requisitos técnicos (KATZ et al., 2019; MORAGA et al., 2017).

Os engenheiros de *software* devem disseminar conhecimento sobre *software* sustentável, orientar as escolhas e tomadas de decisões relacionadas a arquitetura e qualidade que englobam os impactos às três dimensões da sustentabilidade supracitadas. Os envolvidos na definição do produto são responsáveis por optar pelos itens mais sustentáveis, incluindo a sustentabilidade nos requisitos e monitorando sua execução. A gerência deve garantir o bom gerenciamento de um projeto de *software* sustentável, acompanhar as métricas e a efetividade dos objetivos, e as metas sustentáveis estabelecidas (KATZ et al., 2019; MORAGA et al., 2017).

Os impactos que um *software* sustentável consegue alcançar nas três dimensões da sustentabilidade sugeridas por (AHMAD; HUSSAIN; BAHAROM, 2018; JNR; MAJID, 2017; NAUMANN et al., 2011) são apresentadas a seguir:

- **Dimensão Social**: possuem impacto social positivo quando cumpre o objetivo da sua função para o usuário final, são acessíveis em relação a idiomas, linguagens e formas de transmitir o conteúdo. Assim como quando cumprem os requisitos de segurança e garantem a privacidade e proteção dos dados, podem ser testados sem impedimentos e quando garantem a satisfação do usuário.
- **Dimensão Ambiental**: os impactos ambientais estão relacionados ao consumo de recursos em tempo de execução, manutenção, modificação e portabilidade de infraestrutura. Esses recursos estão relacionados tanto aos equipamentos envolvidos, quanto a recursos energéticos consumidos.
- **Dimensão Econômica**: nesta dimensão alguns fatores encontrados nas dimensões anteriores se repetem, como a eficiência energética decorrente de um *software* com melhor utilização de recursos, gera economia financeira. Um sistema testável, com alto nível de qualidade e confiabilidade se torna tolerante a falhas e por consequência reduz as possibilidades de perdas financeiras que uma falha sistêmica poderia gerar. Por fim, a capacidade de manutenção e portabilidade do *software* contribuem para redução de custos com as modificações deste.

Na dimensão ambiental os pilares do *software* sustentável são eficiência energética, otimização de recursos e perdurabilidade. E dentro desses pilares identificou-se três categorias que agrupam as características, conforme apresentado a seguir (CALERO; MORAGA; BERTOA, 2013):

- **Confiabilidade**
  - **Maturidade**: grau de confiabilidade necessário para garantir que um sistema opere com alta disponibilidade.
  - **Disponibilidade**: capacidade do *software* estar disponível e acessível para utilização da operação normal.
  - **Tolerância a falhas**: capacidade de continuidade da operação do sistema independente da presença de falhas, resiliência e redundância.
  - **Capacidade de recuperação**: é a capacidade que um sistema tem de se recuperar em um momento de interrupção de forma rápida e transparente para mitigar os impactos à operação.
- **Capacidade de manutenção**

- **Modularidade**: é o grau de composição do sistema em componentes, de forma que as mudanças sofridas em um componente seja isolada e não afete as demais partes do sistema.
- **Reusabilidade**: é a capacidade de poder reutilizar partes do sistema, sem a necessidade de recriá-las ou duplicá-las.
- **Capacidade de Análise**: é a capacidade de rastrear incidentes de *software* e facilitar a compreensão e identificação de diagnósticos para os problemas encontrados no sistema.
- **Capacidade de Modificação**: é a capacidade que o sistema tem de ser alterado de forma efetiva e eficiente, sem causar novos defeitos.
- **Capacidade de Ser Testável**: é a capacidade de se estabelecer critérios de testes para o sistema, produto ou componente, considerando que a sua realização tenha alta disponibilidade para execução de forma efetiva e eficiente, retratando os cenários e comportamentos do sistema.

- **Portabilidade**
  - **Capacidade Adaptação**: se refere a capacidade de se adaptar a novos ambientes de infraestrutura, utilização e integração com outros *softwares*.
  - **Capacidade de Instalação**: é a capacidade de ser instalado e reinstalado com sucesso em determinado ambiente.
  - **Capacidade de Substituição**: é a capacidade de ser substituído por outro *software* com mesmo propósito.

2.4 Desenvolvimento de *Software* e Normativas

A ISO (*International Organization for Standardization*) e IEC (*International Electrotechnical Commission*) são organizações internacionais de padronização que disponibilizam às organizações conhecimento e padrões internacionais que auxiliam as empresas nas atividades produtivas, de gerenciamento de processos, entre outras.

A série normativa ISO/IEC 33000 dispõe sobre padrões de qualidade para o processo de desenvolvimento de *software*. Os frameworks e processos utilizados para medir a qualidade do processo de desenvolvimento pode ser adaptado para medir o nível de sustentabilidade atingido por um *software* sustentável (LAMI; FABBRINI; BUGLIONE, 2014).

O padrão ISO/IEC 12207 estabelece processos e atividades para todas as etapas do ciclo de vida do desenvolvimento de sistemas, ele serve para definir, monitorar e aperfeiçoar processos de qualidade deste (SINGH, 1996).

A normativa ISO/IEC 15504 oferece um modelo de avaliação dos processos de qualidade do ciclo de desenvolvimento de *software* e foi substituída pela supracitada série da ISO/IEC 33000 (LAMI; FABBRINI; BUGLIONE, 2014).

A NBR (Norma Técnica Brasileira) ISO 14040 especifica princípios e requisitos para utilização do mecanismo avaliação do ciclo de vida (ACV), a fim de levantar os potencias impactos ambientais e a saúde humana, excluindo-se os impactos sociais (ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS - ABNT, 2014).

A ISO/IEC 42010 estabelece padrão para arquitetura de *software* no contexto de ciclo de vida e manutenção do desenvolvimento, com um *framework* que define desde comunicação e planejamento da arquitetura, até a avaliação e manutenção do produto (STANDARDIZATION; NORMALISATION, 1987).

O padrão ISO/IEC/IEEE 24748 fornece um guia consolidado de gerenciamento do ciclo de vida do *software* a partir dos processos definidos na normativa ISO/IEC/IEEE 12207 é destinado aos envolvidos no planejamento da engenharia em todas as etapas do seu ciclo de vida (DUTIL, 2010).

A ISO/IEC 25010:2011 é a norma que define as características necessárias em um *software* para que este seja entregue com qualidade. Esta norma traz modelos de avaliação da qualidade e possui oito características para se alcançar alto nível de qualidade: adequação funcional, eficiência de desempenho, compatibilidade, usabilidade, confiabilidade, segurança, manutenção e portabilidade (ISO/IEC 25010, 2011).

2.5 Qualidade de Software

A qualidade de um *software* se define pelo grau de conformidade que este tem em relação aos requisitos. Mas esta é uma definição simplista que se resume em verificar apenas se o *software* funciona de acordo com o esperado ou não. Além dessa premissa fatores como a

arquitetura, a forma como foi desenvolvido e o modo como ele é gerenciado também compõem a sua qualidade (LEPMETS; RAS; RENAULT, 2011; SHEPPERD, 1990).

O impacto do nível de qualidade de *software* na produtividade do desenvolvimento é direto. Uma vez que um *software* possui uma arquitetura resistente a falhas com menos erros, os profissionais da área consequentemente têm mais tempo para desenvolver novos produtos e fazer melhorias nos já existentes (BHARATHI; SELVARANI, 2015; JAAFAR et al., 2017; SHEPPERD, 1990).

Um código fonte escrito de forma clara e intuitiva, seguindo padrões conhecidos no mercado, possibilita que novos desenvolvedores compreendam o sistema mais facilmente, e assim adquiram velocidade no desenvolvimento de novas demandas (ARCELLI; DI POMPEO, 2017; GAO; PAN; SUN, 2009).

Os requisitos de sustentabilidade dentro de um sistema além de ser um requisito não funcional, podem ser considerados como uma característica de qualidade. O desenvolvimento de *softwares* sustentáveis e os modelos de avaliação da qualidade possuem fatores comuns como objetivo de avaliação, tal como o desempenho deste, que na avaliação da sustentabilidade está relacionada a dimensão ambiental com a eficiência energética. Outros indicadores relevantes em ambas as abordagens são a funcionabilidade, usabilidade, reusabilidade e manutenção. Dessa forma nota-se que existe uma intersecção entre os indicadores de qualidade e de sustentabilidade do *software* porém não necessariamente com os mesmos resultados esperados (GARCÍA-MIRELES et al., 2018).

2.6 Desenvolvimento de *Software* e Métodos Ágeis

Os métodos ágeis tem por princípio a comunicação direta e constante, a produção de código testável, conta a participação ativa do cliente e/ou requisitante, a flexibilidade para tratar as mudanças de escopo, além de contar com equipes pequenas, capacitadas e auto gerenciadas (COCKBURN; HIGHSMITH, 2001)

O manifesto ágil é um documento que contém a definição de quatro valores e doze princípios para o desenvolvimento ágil de sistemas. Várias metodologias de desenvolvimento ágil permeiam esses valores e princípios, as principais características dessas metodologias são a entrega contínua por meio de iterações curtas, a alta flexibilidade para absorver mudanças no

escopo e a proximidade entre as equipes técnicas e o cliente. O manifestos ágil surgiu em 2001, da conversão de metodologias de programação como XP (*Extreme Programming*), método Scrum entre outras, e foi criado por dezessete signatários representantes das metodologias ágeis mais utilizadas na época (HAZZAN; DUBINSKY, 2014).

As principais práticas do *Extreme Programming* (XP) são – programação em pares, jogo do planejamento, pequenas versões e testes de aceitação, e durante a execução do projeto algumas práticas adicionais são requeridas – integração contínua, propriedade do código da equipe e padrão de codificação, comunicação, simplicidade, *feedback* e coragem, que também são considerados como atributos das equipes ágeis (LINDSTROM; JEFFRIES, 2004).

O Scrum utiliza-se de equipes pequenas, requisitos instáveis/mutáveis e iterações curtas, e divide-se em três fases principais – o **pré-planejamento** onde os requisitos são descritos no backlog, são priorizados, a equipe é definida, a ferramenta é escolhida, e a arquitetura é proposta; o **desenvolvimento** é realizado em ciclos, e cada ciclo é composto pelas etapas de análise, projeto, implementação e teste; o **pós-planejamento** que inclui a demonstração do software, integração, teste final e documentação (SOARES, 2004).

Fowler (2004) cita que o XPE elaborado por Kent apresenta quatro critérios para elaborar um sistema simples – roda todos os testes, revela toda intenção, sem duplicação, menor número de classes ou métodos, e conclui inferindo que o design evolutivo deve possibilitar os ajustes durante o projeto e ser maleável, com códigos limpos e reversíveis. De outro lado, Soares (2004) propõe que o uso do XP deve ser direcionado ao desenvolvimento do software e o SCRUM aplicado para gerenciar o projeto de desenvolvimento, uma vez que ambos compartilham os mesmos princípios de agilidade e simplicidade, e são complementares na visão do autor. (FOWLER, 2004; SOARES, 2004)

As metodologias ágeis objetivam a redução de custos e melhoria no desempenho, quando é necessário incorporar mudanças inesperadas nos projetos de desenvolvimento de *software*. Além disso o método trabalha a partir de boas práticas e princípios nos processos, e na comunicação para alcançar o objetivo definido com agilidade (GILL; HENDERSON-SELLERS; NIAZI, 2018; VALLON et al., 2018).

2.7 Programação Orientada a Objetos

A POO (Programação Orientada a Objetos) é um paradigma de programação de alto nível, o que significa que a forma de executar o desenvolvimento está mais próxima aos comportamentos humanos do que das máquinas, sendo oposto ao paradigma de programação estruturada, que é de baixo nível, onde a forma como seu código é escrito possui maior proximidade do entendimento das máquinas (FERNANDES; WERNER, 2019).

Outras diferenças entre a POO e a programação estruturada é que a última é escrita de forma sequencial e com variáveis globais, enquanto o conceito da POO evita o uso de variáveis globais e o seu código pode ser executado simultaneamente, além de possibilitar o reaproveitamento de código quando comparado as linguagens que seguem o paradigma de programação estruturada (DEITEL; DEITEL, 2013).

2.7.1 Conceitos

A POO possui conceitos chave que são Objetos, Classes, Atributos e Métodos conforme detalhado a seguir (DEITEL; DEITEL, 2013; FERNANDES; WERNER, 2019):

- **Objetos**: São as representações das entidades de um sistema, instâncias de uma classe que possuem um estado por onde sistemas desenvolvidos com o paradigma da POO ocorrem.

- **Atributos**: Também conhecidos por propriedades, são as características de um objeto. Variáveis ou constantes que armazenam valores do estado do objeto.

- **Métodos**: ou funções são as ações que um objeto executa, e representam o comportamento dos objetos.

- **Classes**: Podem ser entendidas como contratos de um objeto. As classes dizem como serão as características de um objeto, quais são os seus atributos ou propriedades, e os seus comportamentos ou métodos.

2.7.2 Pilares

Na POO existem quatro pilares básicos a Abstração, a Herança, o Encapsulamento e o Polimorfismo (DEITEL; DEITEL, 2013; FERNANDES; WERNER, 2019):

- **Abstração**: é a identidade única de um objeto com suas determinadas propriedades definidas.

- **Herança**: é um mecanismo que possibilita o reuso de código ao fazer uma classe derivar de outra classe base e herdar as suas características e comportamentos.

- **Encapsulamento**: objetiva controlar o acesso aos atributos e métodos de uma classe. Este pilar de segurança da POO estabelece que quando algum comportamento do sistema é encapsulado em um método, os demais componentes do sistema não têm conhecimento de comportamento acontece, porque somente precisam saber como invocá-lo.

- **Polimorfismo**: é a capacidade de uma classe derivada sobrescrever o comportamento de uma classe base, com isso quando invocado o comportamento ou método não requer o tratamento das diferenças que existem entre classe base e classe derivada, provenientes do conceito supracitado de herança, uma vez que essas diferenças são definidas e tratadas pelo polimorfismo.

Alguns dos pilares e conceitos da programação orientada a objetos favorece a sustentabilidade do *software*, como o encapsulamento que facilita o desenvolvimento, a capacidade de manutenção e portabilidade de uma aplicação. A instanciação de objetos, o polimorfismo e a herança beneficiam o reuso de código (NIERSTRASZ, 1989).

2.8 Design Patterns

Design Patterns ou Padrões de Projetos são boas práticas para soluções de problemas e paradigmas conhecidos na programação de sistemas. Os *Design Patterns* tornaram-se conhecidos após a publicação do livro Padrões de Projeto — Soluções Reutilizáveis de *Software* Orientado a Objetos, escrito em 1994 por quatro engenheiros de *software* que instituíram um vocabulário comum para tratar os projetos de software (GAMMA, Erich. **Design patterns:**

elements of reusable object-oriented software. Pearson Education India, 1995). Nesta obra os autores listaram os problemas mais comuns identificados na programação, e chegaram a vinte e três modelos de solução que tornaram-se os padrões mais utilizados e também serviram de base para a criação de outros padrões (ERIC FREEMAN, ELISABETH FREEMAN, BERT BATES, 2013).

A adoção de padrões de projetos possibilita aos desenvolvedores aumento na produtividade, dado que por utilizar uma solução pronta e já testada no mercado, aprimora-se a comunicação e o compartilhamento do conhecimento sobre o sistema. Observa-se ainda que uma vez que o padrão de desenvolvimento utilizado em um sistema é conhecido, se torna mais fácil a sua compreensão (BOWLES, 2004).

Os padrões de projeto foram desenvolvidos com foco em reutilização de código, desacoplamento e coesão, fatores esses diretamente ligados a agilidade de manutenção e qualidade do código. Os padrões também definem a organização do código, o que pode contribuir para o melhor entendimento do sistema por novos desenvolvedores (BOWLES, 2004; NOUREDDINE; RAJAN, 2015; PRECHELT et al., 2002; YACOUB; AMMAR, 2000).

2.9 *Clean Code* e *Software* Ágil

O termo *Clean Code* ou Código Limpo surgiu em 2008 no livro de mesmo nome, escrito por Robert Martin que se autointitula como "*Uncle Bob*". O autor trabalha com desenvolvimento de *software* desde os anos de 1970 e é um dos signatários envolvidos na criação do Manifesto Ágil em 2001. Em seu livro *Clean Code* ele apresenta suas percepções de como deve ser escrito e mantido um bom código fonte em forma de padrão de desenvolvimento, tornando-se referência na comunidade de profissionais da área (MARTIN, 2011).

O *Clean Code* aborda sobre a importância de se criar um código legível que seja explicável *per se*, sem a necessidade de uma documentação ou comentário para o entendimento de outro profissional. O autor defende que métodos e funções devem ser simples e objetivos, com uma única responsabilidade, e que os trechos de código nunca devem ser repetidos, porque código duplicado além de dificultar o entendimento, dificulta a manutenção do mesmo (MARTIN, 2011).

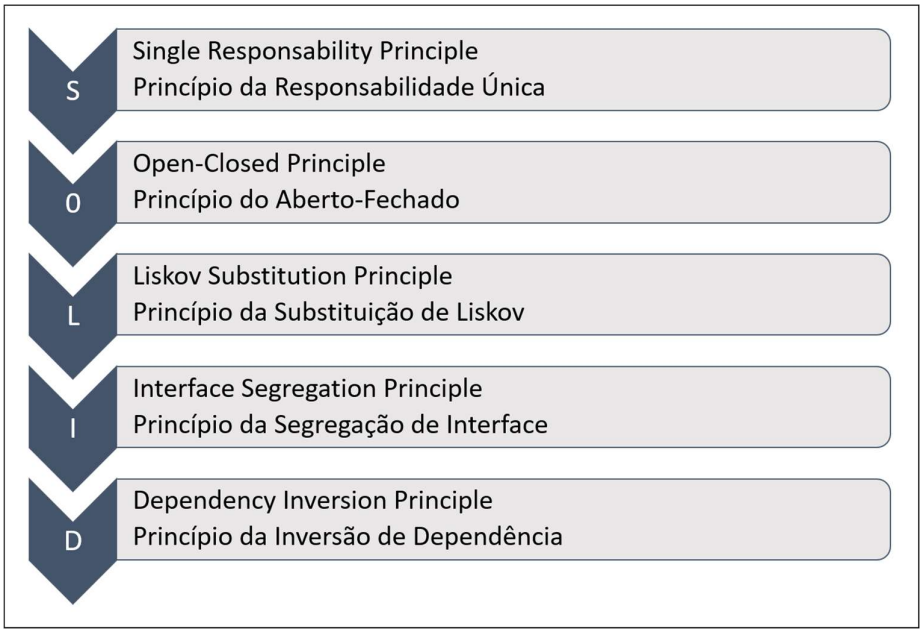
Um dos princípios deste padrão de desenvolvimento é que código não deve ser rígido, deve estar aberto à mudanças, conceito comum nas metodologias ágeis, mas ao mesmo tempo deve ser fechado para alterações (LATTE; HENNING; WOJCIESZAK, 2019; MARTIN, 2011).

Este padrão de desenvolvimento além de focar na legibilidade do código, também direciona para redução do tempo de manutenção, o que implica no aumento de produtividade. Conforme Martin (2011) muitas vezes a produtividade da programação é perdida devido à complexidade de um código "ruim". O *Clean Code* também esclarece sobre a necessidade de validação realizada com testes limpos, princípio este diretamente ligado a qualidade do *software* (GUPTA et al., 2016; LATTE; HENNING; WOJCIESZAK, 2019; MARTIN, 2011).

2.10 SOLID

O S.O.L.I.D é um acrônimo dos cinco princípios da programação orientada a objetos (Figura 1) criado por Michael C. Feathers (2004) ao reorganizar estes princípios em siglas. O autor criou o acrônimo a partir da primeira letra de cada um dos cinco primeiros princípios da programação orientada a objetos e *design* desenvolvido por Robert Martin, autor da obra *Clean Code*. Os cinco princípios são (MARTIN, 2014; OKTAFIANI; HENDRADJAYA, 2018):

Figura 1 - Princípios do SOLID



Fonte: elaborado pela própria autora.

2.10.1 Single Responsability Principle

O *Single Responsability Principle* (SRP) ou Princípio da Responsabilidade Única é o fundamento cuja premissa estabelece que cada classe ou componente de um sistema, deve ter uma única necessidade de existir e ser responsável por apenas uma tarefa. Dessa forma se evita a criação das chamadas "classes deus" como citado por Martin (2011), classes que fazem tudo dentro de um sistema, com alto acoplamento e baixa coesão.

A existência de classes com alto acoplamento e baixa coesão, em um sistema, dificulta a manutenção e estabilidade dele, porque ao promover uma alteração em uma parte do sistema acaba-se comprometendo outra parte que não deveria ser afetada. Outro problema relacionado ao alto acoplamento e baixa coesão está relacionado as limitações no reaproveitamento do código. (RODRIGUES; SOUZA; FIGUEIREDO, 2014)

Como resultado da implementação deste princípio, obtém-se classes mais coesas e menos acopladas, que são mais simples de testar. Esse fator melhora a qualidade do código, reduz o custo de manutenção e aumenta a produtividade do desenvolvimento.

2.10.2 Open-Closed Principle

O *Open-Closed Principle* (OCP) ou Princípio Aberto-Fechado conceitua que os componentes de um sistema devem estar abertos para extensões e fechados para alterações. Este princípio está ligado aos conceitos do Manifesto Ágil onde as mudanças no sistema são bem-vindas, porém para serem realizadas de forma segura o código deve estar fechado para alterações, mas permitir a sua extensão para criação de novos comportamentos. Adotando-se este princípio, a realização de uma alteração no sistema vai gerar uma nova extensão do comportamento anterior e este não será alterado.

O resultado obtido com a adoção deste princípio é a maior facilidade para absorver alterações sem comprometer a qualidade e estabilidade do código, tendo com a premissa o uso adequado das abstrações previstas nos princípios apresentados a seguir: LSP (Liskov Substitution Principle), ISP (Interface Segregation Principle) e DIP (Dependency Inversion Principle).

2.10.3 Liskov Substitution Principle

O *Liskov Substitution Principle* (LSP) ou Princípio de Substituição de Liskov resgata um conceito defendido por Barbara Liskov em 1987 com uma fórmula, que de forma resumida, prega que todos os objetos derivados de uma abstração podem substituir a sua abstração sem alteração no comportamento. Trazendo essa ideia para o SOLID na orientação a objetos, a não violação do LSP implica no uso mais seguro do polimorfismo. Para exemplificar este conceito, pode-se dizer que se uma "classe base" possui um método que retorna como resultado um número inteiro, as suas "classes derivadas" não devem retornar algo diferente de um número inteiro neste mesmo método.

Dessa forma pode-se referenciar tanto uma abstração ou derivação da classe base, quanto a própria classe base sem comprometer o comportamento esperado. A não violação do princípio LSP implica diretamente na qualidade do código, porque minimiza a ocorrência de erros inesperados.

2.10.4 Interface Segregation Principle

O *Interface Segregation Principle* (ISP) ou Princípio da Segregação de Interface defende que é melhor para um sistema escrito em linguagem orientada a objetos a criação de interfaces específicas ou coesas do que interfaces genéricas com muitas responsabilidades. A implementação deste princípio está diretamente ligada a não violação do princípio LSP, visto que o uso de interfaces genéricas pode fazer com que não se tenha efetivação em todos os métodos da interface, gerando um comportamento inesperado que o LSP prevê evitar.

O resultado do uso deste princípio, assim com o LSP, está relacionado a qualidade do código ao evitar erros inesperados decorrentes de derivações sem a implementação adequada, e na facilidade da manutenção de um código mais coeso.

2.10.5 Dependency Inversion Principle

O *Dependency Inversion Principle* (DIP) ou Princípio da Inversão de Dependência trata de não depender de classes instáveis, mas sim das suas abstrações. Dessa forma uma classe não precisa conhecer as classes derivadas, apenas a sua classe base. Removendo essa

responsabilidade de uma classe, a sua portabilidade para outro projeto ou sistema fica mais simples, visto que ele terá menos dependências para serem levadas com abstração da classe.

O cumprimento deste princípio é parte do objetivo de desacoplamento das classes, trazendo impacto direto na manutenção, produtividade e qualidade do código.

2.11 DDD

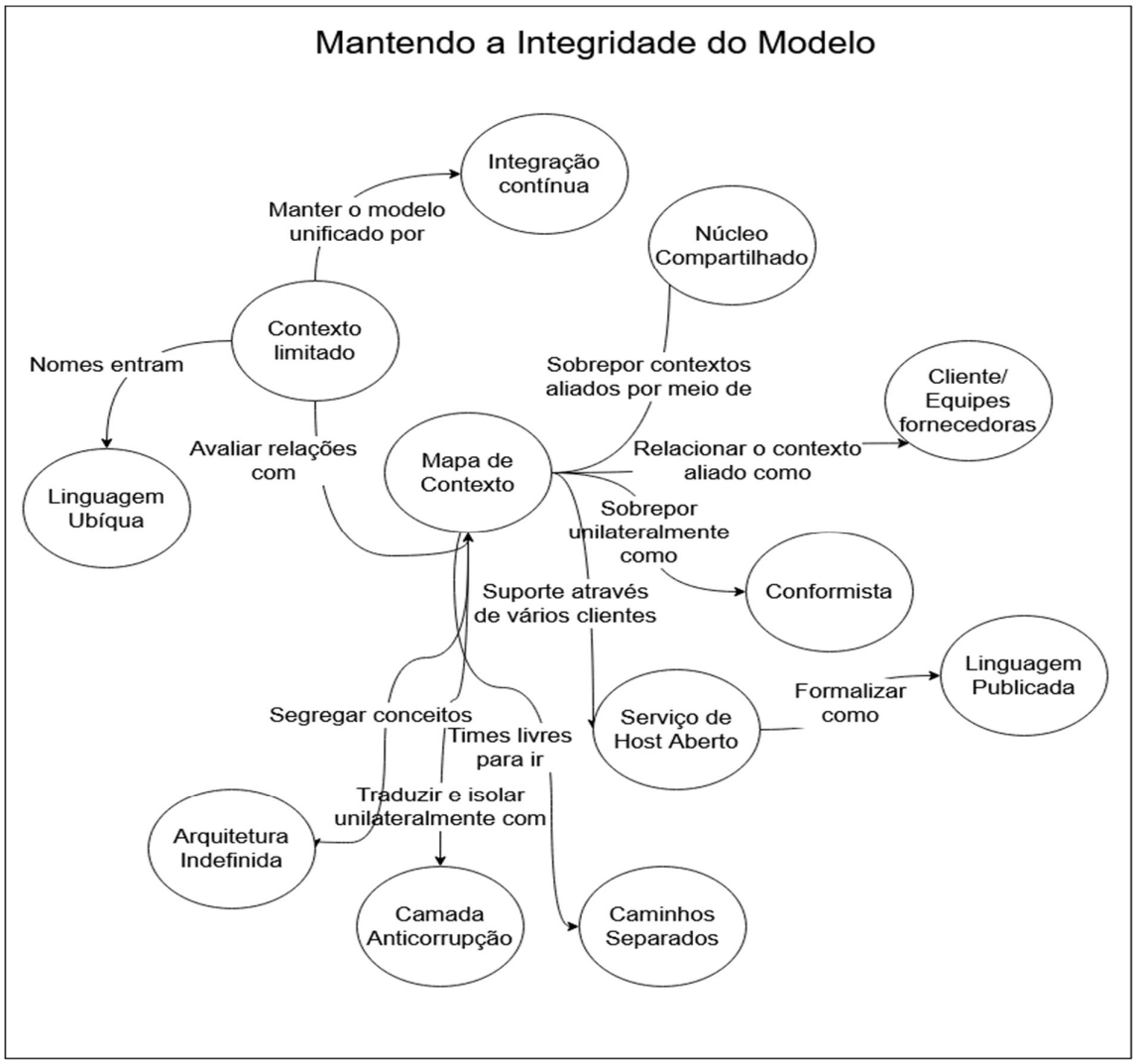
O *Domain-Driven Design* (DDD) ou Projeto Orientado à Domínio é um conjunto de boas práticas e padrões de modelagem e desenvolvimento aplicáveis ao paradigma de programação orientada a objetos (EVANS, 2010; LE; DANG; NGUYEN, 2017).

Este conceito defende a ideia de que a modelagem, tanto do banco de dados quanto do código fonte, deve estar muito próxima ao modelo do negócio. Considerando esta condição é extremamente necessário que especialistas de negócios e especialistas de tecnologia tenham uma linguagem única sobre o modelo do negócio. Com todos os envolvidos utilizando uma mesma linguagem, cientes do mesmo significado para cada termo empregado, os erros de sistemas decorrentes desse tipo de desinformação tendem a ser minimizados (EVANS, 2010; RADEMACHER; SORGALLA; SACHWEH, 2018).

O DDD não prega boas práticas apenas para o desenvolvimento do *software* em si, mas para outras fases do ciclo de vida, com padrões de comunicação, testes e processos de trabalho, objetivando reduzir os riscos com falhas de sistema e dificuldades de manutenção (EVANS, 2010; RADEMACHER; SORGALLA; SACHWEH, 2018; SMITH; JEGATHEESAN; KELLY, 2017).

A figura 2 exibe os princípios da estratégia do DDD para se manter a integridade de um modelo conceitual único.

Figura 2 - DDD - Como manter a integridade do modelo



Fonte: Wikipedia. Disponível em: https://en.wikipedia.org/wiki/Domain-driven_design (adaptado e traduzido pela autora)

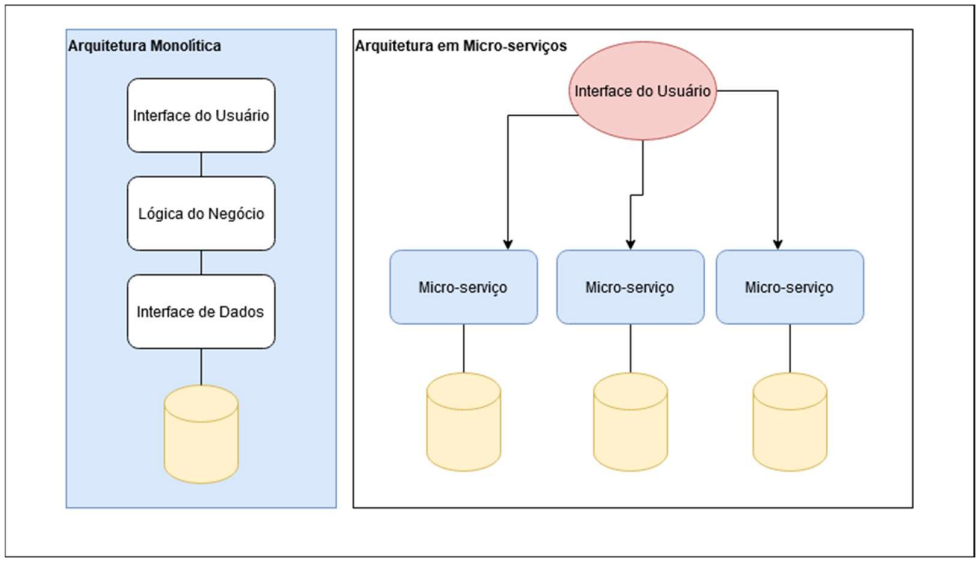
2.12 Micro-serviços

Micro-serviço é um padrão de arquitetura de *software* que divide as funcionalidades dos sistemas em pequenos serviços geralmente expostos em API[2] (*Application Programming*

[2] API – siga em inglês e em língua portuguesa é traduzida como Interface de Programação de Aplicação. como interfaces legíveis por máquina que conectam vários aplicativos, governam a interação de aplicativos e eliminam

Interface) e se comunicam através do protocolo de comunicação HTTP[3] (*Hipertext Transfer Protocol*) (PAUTASSO et al., 2017; SOMMERVILLE, 2011). A figura 3 expõe sua diferença em relação a arquitetura monolítica.

Figura 3 - Arquitetura em Micro-serviços versus Arquitetura Monolítica



Fonte: elaborado pela própria autora.

Este tipo de arquitetura permite que as funcionalidades do sistema sejam independentes, ou seja, uma funcionalidade pode ser alterada ou publicada sem afetar as demais. Assim, as aplicações se tornam mais escaláveis e resistentes a falhas, dado que uma funcionalidade, mesmo que apresente falhas, não compromete o sistema inteiro (RADEMACHER; SORGALLA; SACHWEH, 2018; SOMMERVILLE, 2011; TUSJUNT; VATANAWOOD, 2018).

As equipes de desenvolvimento também ganham agilidade e velocidade na entrega, já que aplicações menores tendem a ser menos complexas do que aplicações grandes, e mais pessoas conseguem trabalhar em serviços diferentes sem afetar outras demandas. Além disso,

---

a necessidade de conhecer o funcionamento interno de como a funcionalidade de uma API é fornecida (Jacobson, D.; Brail, G.; and Woods, D. APIs: A Strategy Guide. Sebastopol, CA, USA: O'Reilly,2011).

[3] HTTP em português Protocolo de Transferência de Hipertexto refere-se a um protocolo de comunicação utilizado por sistemas de informação para transferência de dados, sendo este mecanismo a base das comunicações da Word Wide Web (BERNERS-LEE, Tim; FIELDING, Roy; FRYSTYK, Henrik. Hypertext transfer protocol--HTTP/1.0. 1996.)

os profissionais ganham a flexibilidade de poder utilizar a tecnologia mais adequada para cada serviço (BOGNER et al., 2018; PAUTASSO et al., 2017; RADEMACHER; SORGALLA; SACHWEH, 2018).

2.13 Git Flow

O Git é um sistema de controle de versões criado por Linus Torvalds para o desenvolvimento do sistema operacional Linux. Com ele é possível fazer controle do histórico de versões de um código fonte, evoluir e voltar a qualquer ponto do histórico dos arquivos pertencentes a um repositório Git (GERMAN; ADAMS; HASSAN, 2015).

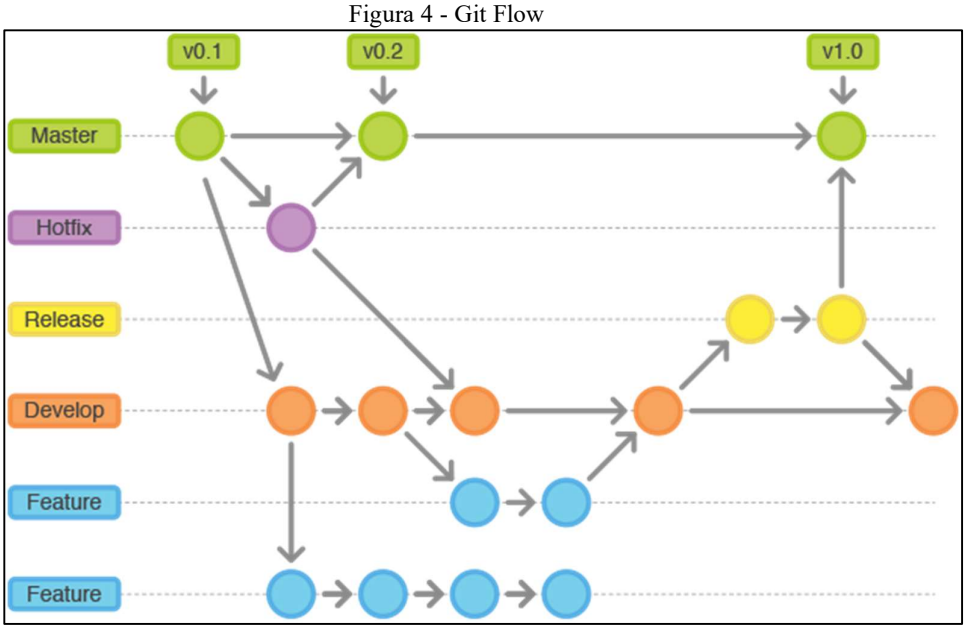
Por sua vez, o *Git Flow* é um conjunto de boas práticas e diretrizes, que auxiliam as equipes de desenvolvedores na adoção do Git, além de ser uma ferramenta *open source* que se integra ao Git, para facilitar a utilização dos padrões de nomenclatura e trabalho. O sistema de controle de versões Git utiliza ramificações do código fonte conhecidas como *branches*. Essas ramificações são cópias da versão original dos códigos fonte sobre o qual pode-se fazer modificações sem alterar a ramificação original, uma vez que cada ramificação possui uma linha histórica de controle de versões (CHANG; OU; DENG, 2019; GERMAN; ADAMS; HASSAN, 2015).

Na figura 4 mostra-se as ramificações representadas pelas linhas horizontais, e os seus respectivos tipos representados pelas caixas a esquerda. Estes tipos correspondem às nomenclaturas padrões do *Git Flow*, conforme detalhado abaixo (CHANG; OU; DENG, 2019):

- ***Master***: é a ramificação do código oficial que geralmente já está no ambiente produtivo.
- ***Hotfix***: é utilizada para correções pontuais do ambiente de produção, o seu código fonte sempre se origina da última versão da *Master*.
- ***Release***: concentra um conjunto de funcionalidades desenvolvidas que estão prontas para migrarem para a ramificação *Master* do repositório.
- ***Develop***: é o código de onde partem as novas funcionalidades e concentra as modificações da próxima versão do sistema, o seu próximo passo é ser mesclado com a *Release* e finalmente com a *Master*.

- ***Feature***: são *branches* especificas para o desenvolvimento de novas funcionalidades. O seu código se origina da última versão da *Develop,* e após finalizada suas alterações são migradas para esta última, e a partir dessa seguem o fluxo até chegar a *Master*.

Figura 4 - Git Flow



Fonte: iMasters. Disponível em: https://imasters.com.br/agile/fluxo-de-desenvolvimento-com-gitflow

A utilização do modelo *Git Flow* em pequenas equipes de desenvolvedores, pode parecer desnecessária devido a quantidade de etapas e diretrizes a serem cumpridas. Todavia, em equipes grandes e descentralizadas ou remotas o modelo proporciona redução de riscos segurança e melhora a qualidade do código. Este ganho torna-se importante uma vez que um dos maiores problemas no gerenciamento de controle de versões de código são decorrentes da mesclagem de modificações de diversos desenvolvedores, realizadas de maneira incorreta, e que ao se sobrepor impactam negativamente no sistema (CHANG; OU; DENG, 2019; SAITO et al., 2016).

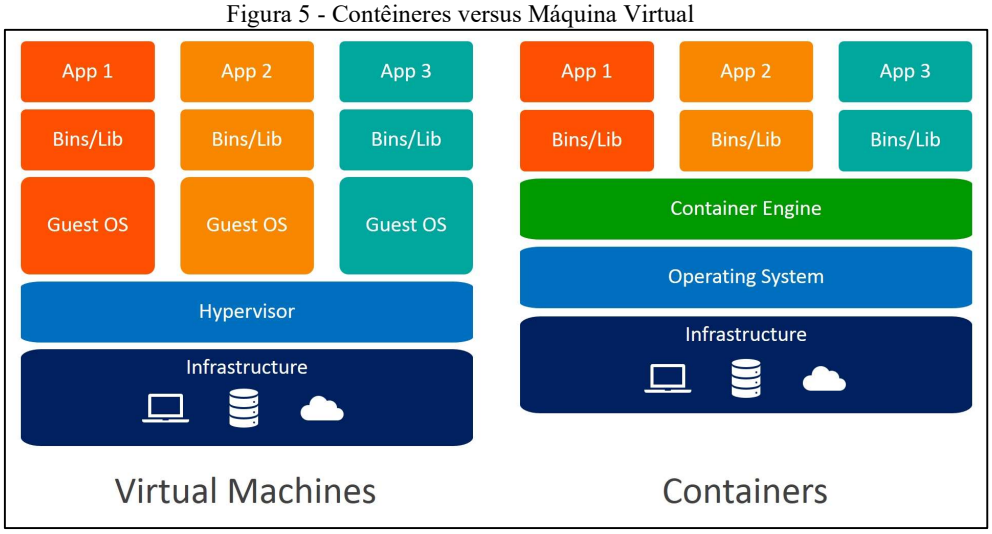
2.14 Contêiner

O contêiner de acordo com a definição de DOCKER "...é uma unidade padrão de software que empacota o código e todas as suas dependências para que o aplicativo seja executado de

forma rápida e confiável, e de um ambiente de computação para outro"[4]. Trata-se ainda de uma tecnologia diferente das conhecidas máquinas que virtualizam sistemas operacionais. Os contêineres executam diversas aplicações de forma isolada, compartilhando recursos como sistema operacional e memória da mesma máquina (LINGAYAT; BADRE; GUPTA, 2018).

Essa tecnologia tem se mostrado uma alternativa a virtualização de sistemas operacionais, dado que viabilizam a redução de custos com manutenção ao compartilhar o mesmo sistema operacional, e conferem maior velocidade de implantação e recuperação, além de serem portáveis. (CITO; GALL, 2016; LINGAYAT; BADRE; GUPTA, 2018; MADHUMATHI, 2018).

Figura 5 - Contêineres versus Máquina Virtual



Fonte: WeaveWorks. Disponível em: https://www.weave.works/blog/a-practical-guide-to-choosing-between-docker-containers-and-vms

A figura 5 mostra como funciona a tecnologia de contêineres e quais são as diferenças ao ser comparada com as tecnologias de máquina virtual. Nela pode-se ver que na virtualização cada aplicação possui o seu próprio sistema operacional sob a mesma infraestrutura ou máquina física, e quem faz o controle é o *Hypervisor*, um processo que cria e executas as máquinas virtuais. Já na tecnologia de contêineres as aplicações além de compartilhar a mesma

[4] A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another.

infraestrutura também compartilham o mesmo sistema operacional e a *Container Engine* controla os contêineres que empacotam a aplicação e suas dependências.

2.15 Trabalhos correlatos

O estudo de Procaccianti, Fernández e Lago (2016) sobre a eficiência energética de um *software*, realizado por meio da comprovação experimental, mostra que o consumo energético tem adquirido importância no desenvolvimento. Neste trabalho os autores compararam o desempenho energético de *softwares* com funcionalidades similares fazendo uso de diferentes arquiteturas, com o objetivo de obter um guia de melhores práticas.

O artigo de Kern et al. (2018) apresenta os resultados de um projeto de *Design* de *Software* Sustentável encomendado pela Agência Federal Alemã de Meio Ambiente. O projeto usa como base projetos anteriores para fazer comparações entre aplicações similares em relação ao seu desempenho e consumo energético. A finalidade dessa pesquisa é estabelecer critérios para classificar um *software* como sustentável. O intuito é que esses critérios possam ser utilizados por engenheiros para guiá-los na construção de *softwares* sustentáveis. O artigo também conceitua o termo e levanta questionamentos sobre o que é necessário para obter tal classificação, além de incluir provocações como levantar questionamentos referente a intangibilidade da natureza de um produto de *software* ser suficiente para torná-lo sustentável.

Jaafar et al. (2017) realizaram um estudo empírico com vinte e duas versões de quatro sistemas, a fim de investigar os impactos na qualidade do *software* decorrente das mudanças de padrões de projeto e o uso de anti-padrões. García-Mireles et al. (2018) abordam sobre a interação entre qualidade de *software* e os objetivos da sustentabilidade ambiental, como eles se correlacionam e como podem co-existir no desenvolvimento de uma aplicação.

Os estudos realizados nestes artigos são muito importantes para o presente trabalho porque reforçam a relevância que o desenvolvimento de um *software* tem no aspecto ambiental e o quanto os padrões de desenvolvimento estão ligados a qualidade do *software*, e o quanto que estes por sua vez estão relacionados à sustentabilidade.

Dentre os estudos indicados, dois artigos específicos sobre *software* sustentável se assemelham a este trabalho no tocante ao conceito do *software* sustentável, entretanto diferem no objetivo geral da pesquisa. O artigo de Kern et al. (2018) tem por objetivo apresentar os

critérios para a classificação de um *software* sustentável. Já o artigo de Procaccianti, Fernández e Lago (2016) tem como resultado obter um guia de boas práticas. Por outro lado, o presente estudo foca em identificar quais os princípios de programação atendem ou não a estes critérios e boas práticas apresentados nesta monografia.

3 DESENVOLVIMENTO

3.1 Proposta

A proposta deste trabalho é identificar os padrões e tecnologias de desenvolvimento de *software* atuais e mais notório entre os profissionais da área, e realizar um estudo comparativo entre os resultados esperados ao se empregar tais paradigmas e os resultados esperados de um *software* sustentável, a fim de encontrar os padrões que promovem a sustentabilidade.

Com a realização de uma *survey* mapeou-se os padrões mais utilizados, seus impactos e valores mais relevantes, que foram avaliados pelos profissionais no processo de desenvolvimento. Com a pesquisa também foi possível estimar o quanto os profissionais da área estão familiarizados com a sustentabilidade no desenvolvimento de *software*.

Após a aplicação e análise dos resultados obtidos com a *survey* obteve-se um quadro comparativo entre padrões de desenvolvimento e padrões de *software* sustentável.

3.2 Survey

A pesquisa foi efetuada no período de 20 de maio à 06 de junho de 2020, por meio da plataforma de *survey* Google *Forms*, e contou com a participação voluntária de profissionais de desenvolvimento de *software* sobre temas relacionados a padrões de projetos e sustentabilidade. A divulgação foi realizada por meio de redes sociais para profissionais com os quais a autora já teve contato e com suas respectivas redes de relacionamento profissional.

O total de respondentes da pesquisa foi de cento e um, destes noventa e cinco se autointitularam profissionais ou ex-prossifionais do processo de desenvolvimento de *software*.

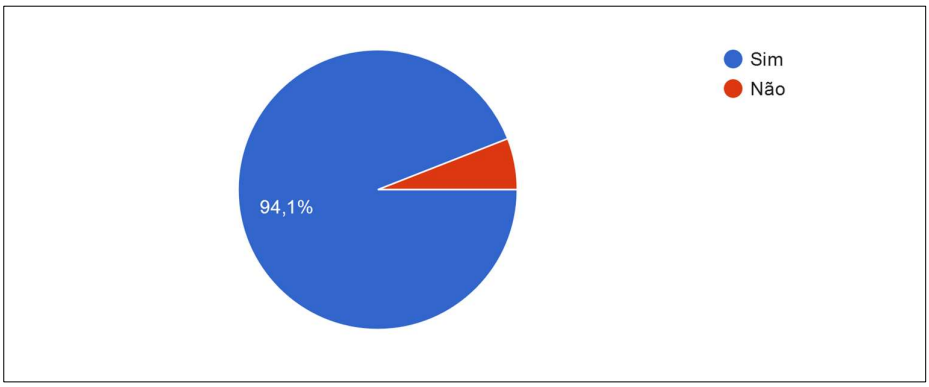
3.2.1 Resultados

A pesquisa foi dividida em quatro seções. A seguir será apresentada cada seção, e como a pergunta foi realizada aos entrevistados, quais eram suas opões de escolhas e métricas utilizadas, seguidas da apresentação dos seus resultados. Para todas as perguntas foram gerados

um ou mais gráficos, e serão mostrados aqui os mais expressivos para a pesquisa. No anexo A pode-se encontrar todos os demais gráficos que não estão apresentados neste item.

I – Você trabalha ou já trabalhou com desenvolvimento de *software*?

Figura 6 - Total de respondentes que trabalham com desenvolvimento de *software*

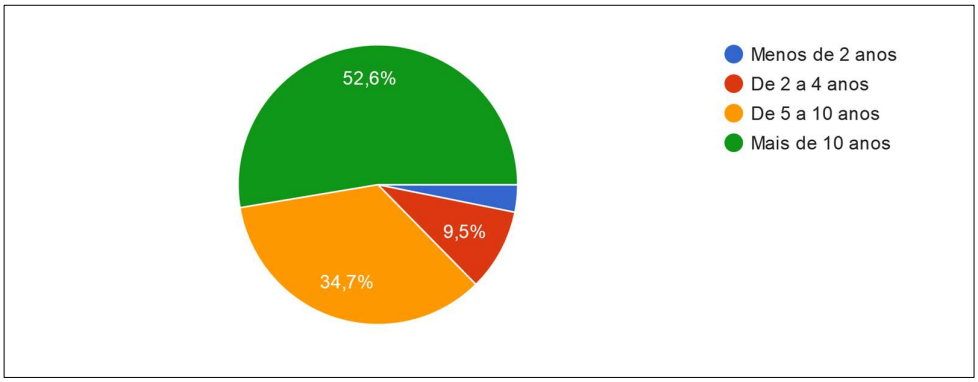


Fonte: Dados da pesquisa (elaborado pela própria autora).

A primeira pergunta da pesquisa serviu como filtro para selecionar as repostas determinantes da pesquisa apenas entre profissionais ou ex-profissionais da área de desenvolvimento de *software*. Dos cento e um respondentes, noventa e cinco ou 94,1% se enquadraram dentro do perfil desejado e continuaram a responder as demais questões da pesquisa.

II – A quanto tempo você trabalha com desenvolvimento de *software*?

Figura 7 - Tempo de experiência com desenvolvimento de *software*



Fonte: Dados da pesquisa (elaborado pela própria autora).

A segunda pergunta teve como objetivo classificar os respondentes em faixas de acordo com o tempo de atuação com desenvolvimento. Conforme gráfico representado na figura 7,

52,6% dos respondentes possuem mais de dez anos de carreira com desenvolvimento de *software,* 34,7% de cinco a dez anos, 9,5% de dois a quatro anos de experiência e os demais menos de dois anos de experiência. Esse resultado demostra que os respondentes da pesquisa em sua maioria possuem alto nível de senioridade na sua função dentro do desenvolvimento de *software*.

III – O quanto você avalia que conhece sobre os itens abaixo:

A terceira seção da pesquisa visava identificar o quanto os respondentes conheciam acerca de determinadas tecnologias ou padrões existentes da programação. Foi utilizada a escala Likert[5] de um a cinco, onde cinco representava muito conhecimento ou familiaridade com o tema e um representava nenhum conhecimento sobre o assunto, conforme mostrado na tabela 1.

Tabela 1 - Resultado da pesquisa sobre conhecimento de padrões e tecnologias

O quanto você avalia que conhece sobre os itens abaixo:	**Escala Likert**					**Total**
	5	**4**	**3**	**2**	**1**	
POO - Programação Orientada a Objetos	45	31	18	1	0	95
SOA - Arquitetura orientada serviços	25	23	29	13	5	95
Micro Serviços	20	13	41	20	1	95
SOLID: Os 5 princípios da POO	15	22	22	18	18	95
Clean Code	25	24	30	9	7	95
GIT Flow	16	18	31	18	12	95
Design Patterns	19	25	34	14	3	95
DDD - Domain-Driven Design	16	12	31	17	19	95
Desenvolvimento de *Software* Ágil	45	21	23	6	0	95
MVC – Multicamadas	35	24	20	9	7	95
Rest / Restfull	28	22	24	10	11	95
Software Sustentável	14	7	21	22	31	95
Contêineres	18	12	30	17	18	95

Fonte: Dados da pesquisa (elaborado pela própria autora).

No Anexo A mostra-se os dois gráficos com os resultados individuais de cada pergunta presente na tabela 1, um apresenta o resultado geral da pesquisa e o outro apresenta o resultado

[5] A escala Likert original é um conjunto de afirmações (itens) oferecidos para uma situação real ou hipotética em estudo. Os participantes são solicitados a mostrar seu nível de concordância (de discordo totalmente a concordo totalmente) com a afirmação dada (itens) em uma escala métrica. Aqui, todas as falas combinadas revelam a dimensão específica da atitude em relação à questão, portanto, necessariamente interligadas entre si. (Joshi et al.; BJAST, 7(4): 396-403, 2015; Article no. BJAST.2015.157)

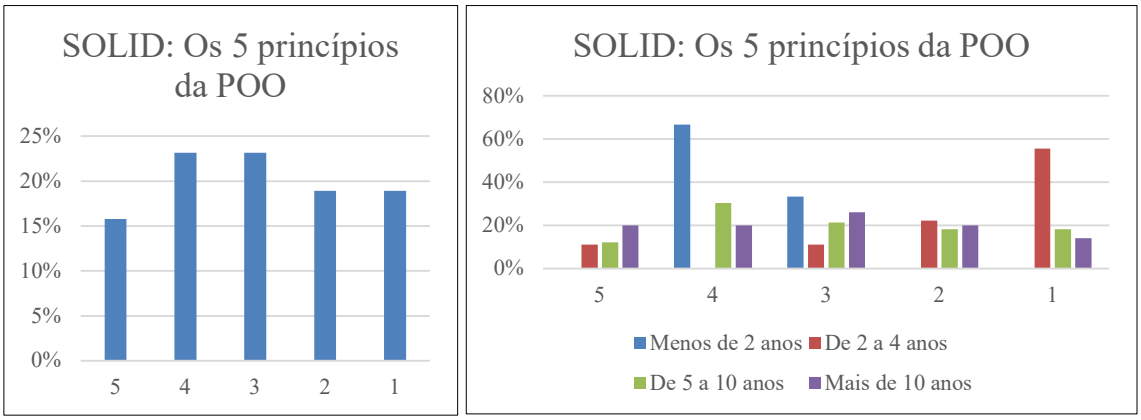
da mesma pergunta por tempo de experiência de acordo com a resposta da pergunta anterior realizada na segunda seção da pesquisa. (Figura 7).

Como pode-se observar na tabela 1 o total de respostas sobre programação orientada a objetos demonstra que grande parte dos pesquisados tem familiaridade em relação a este paradigma da programação. Com este resultado pode-se considerar que o tema é bem difundido entre os profissionais respondentes. A pesquisa revelou ainda que apenas os profissionais com mais de dez anos de experiência responderam conhecer pouco ou nada sobre a Programação Orientada a Objetos. (vide Anexo A)

Quanto a arquitetura SOA a maior parte dos pesquisados demonstrou conhecimento de médio a alto sobre o assunto. Os profissionais com tempo de experiência de cinco anos ou mais demonstram mais domínio sobre esse assunto, enquanto os profissionais com menos de dois anos de experiência, que contou com menor número de respondentes de toda a pesquisa, selecionaram a mesma resposta para a pergunta, classificando o assunto como muito conhecido, porém pouco ou não utilizado.

Em relação a arquitetura de micro serviços todas as faixas demonstraram conhecimento mediano sobre o assunto, o que representa um conhecimento mais teórico do que prático sobre o tema.

Figura 8 - Resultado da pesquisa para SOLID



Fonte: Dados da pesquisa (elaborado pela própria autora).

O gráfico (Figura 8) da pesquisa para o acrônimo SOLID, que representa os cinco princípios da programação orientada a objetos, apresentou resultado indefinido, visto que todas as escalas receberam um número similar de respostas, estando entre 15 e 20 por cento do total.

Os destaques no gráfico ficam com as faixas de experiência de menos de 2 anos e de 2 a 4 anos, onde a primeira teve o maior número de respostas na escala quatro e o segundo na escala um. Devido ao fato do termo ser relativamente recente é aceitável o resultado mostrar que pessoas iniciantes na carreira conhecem mais sobre esse assunto em específico, por possuírem formação acadêmica mais recente.

E quanto ao *Clean Code* de onde o SOLID é derivado, o gráfico tende mais ao grau de familiaridade de médio a alto com 84% das respostas, em que a maioria se concentra na escala três de conhecimento ou familiaridade mediano, conhecimento considerado mais teórico do que prático.

O gráfico das repostas para a pergunta sobre o GIT Flow, padrão de versionamento de código com foco na integração contínua, tem uma curva normal com alto número de respondentes no centro da escala do eixo horizontal. Quando se observa o gráfico especializado por tempo de experiência no ramo de atuação (vide Anexo A), o destaque fica com os profissionais da faixa de menos de dois anos de experiência, que indicaram o nível quatro na escala em suas respostas. Observa-se ainda que este nível não foi selecionado por outras faixas.

Sobre os D*esign Patterns* ou padrões de projetos assim como na pergunta sobre o *Git Flow* o gráfico geral mostra que a maior parte das respostas se concentraram entre o centro da escala e os níveis mais altos, com poucas respostas para os dois níveis mais baixos da escala. Na escala um os únicos respondentes pertenciam a faixa de experiência de dois a quatro anos.

O DDD (*Domain-Driven Design*) metodologia de projeto orientado a domínio teve a maior parte dos respondentes indicando os níveis entre médio a baixo, com pico de respostas na escala três onde os pesquisados com tempo de experiência de mais de cinco anos foram os que mais contribuíram com este resultado (vide anexo A).

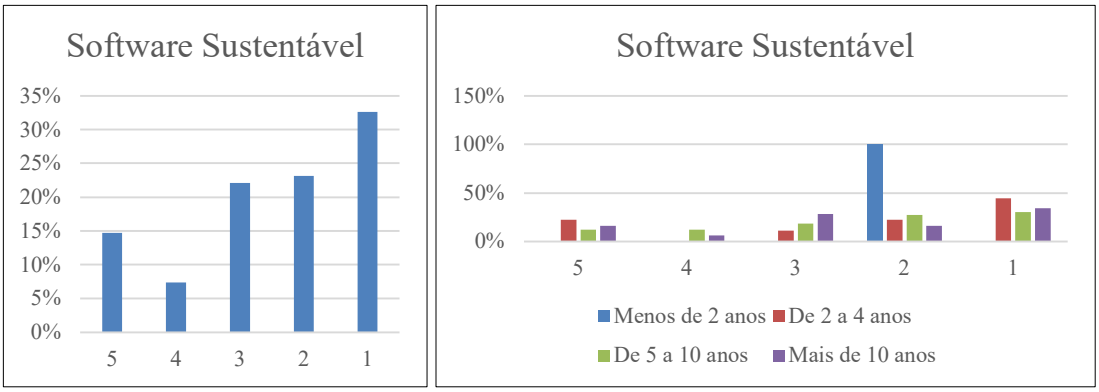
O gráfico das respostas para o conceito de desenvolvimento de *software* ágil mostra que este método está bem difundido entre os pesquisados, já que quase todas as faixas concentraram suas respostas na escala cinco. A exceção foi observada na faixa de dois a quatro anos que registrou a maioria das respostas na escala três. Na escala um que se encaixam as pessoas que não tem nenhum conhecimento ou familiaridade sobre o assunto não houve respostas.

O padrão de desenvolvimento de projetos de *software* MVC ou Multicamadas no geral teve como resultado um gráfico ascendente, onde pouco mais de 15% responderam na escala

um ou dois sobre seu grau de conhecimento do assunto, e a maioria de quase todas as faixas classificaram a pergunta na escala cinco. Observou-se que na faixa de menos de dois anos de experiência concentra a maioria das expostas na escala quatro. Este resultado mostra que o conceito é bem familiarizado entre os profissionais menos experientes.

Os princípios de comunicação entre sistemas, o REST mostra um gráfico de conhecimento com aproximadamente 80% das respostas entre as escalas cinco e três, o que indica forte disseminação do assunto entre os pesquisados.

Figura 9 - Resultado da pesquisa para *Software* Sustentável



Fonte: Dados da pesquisa (elaborado pela própria autora).

Já o *software* sustentável que é o tema principal deste estudo mostra que mais de 30% dos pesquisados não conhecem e ou nunca ouviram falar sobre o assunto. Menos de 15% conhecem bem ou dominam o assunto, e dentre estes, todos tem mais de dois anos de experiência com programação. Destaca-se ainda que o maior número de respondentes está na faixa de dois a quatro anos de experiência. A faixa de menos de dois anos de experiência concentrou as suas respostas na escala dois. Se somarmos os resultados das escalas um e dois temos 56% das respostas, o que mostra que o assunto não está difundido entre os profissionais pesquisados.

O gráfico resultante do conhecimento sobre contêiner mostra que mais de 30% dos pesquisados tem conhecimento mediano sobre o assunto, mas não chegam a empregá-lo na prática. No outro extremo observou-se que 37% dos respondentes indicaram conhecerem muito pouco ou nada sobre o assunto, classificando o assunto nas escalas um e dois.

Aproximadamente 22% dos respondentes com tempo de experiência de mais de dois anos responderam que conhecem bem o assunto e até o utilizam. Um resultado, que no geral,

foi bem equilibrado sobre o tema Contêiner em todas as faixas de experiência profissional dos respondentes.

IV – A quarta seção da pesquisa identifica quais os princípios empregados e as preocupações do profissional de desenvolvimento de *software* ao construí-lo:

Tabela 2 - Resultado da seção IV da pesquisa

Ao decidir utilizar um determinado padrão de desenvolvimento, o quanto você leva em conta os impactos/ganhos abaixo com o que foi desenvolvido:	**Dimensão da Sustentabilidade**	**Escala Likert**				
		5	**4**	**3**	**2**	**1**
Eficiência energética	AMBIENTAL	5	10	22	14	44
Consumo de água		3	5	11	17	59
Desempenho/Processamento		49	24	20	0	2
Impactos ambientais		5	11	16	16	47
Reutilização de código	ECONÔMICO	52	32	8	2	1
Qualidade de código		53	35	6	1	0
Manutenção do código		43	40	12	0	0
Custos		26	30	29	8	2
Redução de despesas		17	36	33	6	3
Impactos sociais	SOCIAL	3	22	26	14	30
Usabilidade		39	43	13	0	0
Entregar valor ao cliente		63	25	7	0	0
Legibilidade do código		48	28	16	2	1

Fonte: Dados da pesquisa (elabora pela própria autora).

Analisando as respostas indicadas e organizando-as dentro das três dimensões da sustentabilidade, observa-se que quanto a eficiência energética apenas 5,3% dos respondentes levantaram este impacto como relevante e presente no seu trabalho ao considerar os impactos de suas atividades, conforme tabela 2. Para 46,3% dos respondentes (distribuídos em todas as faixas de experiencia profissional) afirmaram que nunca pensaram neste impacto ao desenvolver um *software.* A única faixa de experiência que não teve a maioria das respostas na escala 1 foi a faixa de dois a quatro anos de experiência que concentrou pouco mais de 50% das suas respostas na escala 3.

Já em relação a reutilização de código 88,4% dos pesquisados pontuaram o tema nas escalas cinco ou quatro, demonstrando que essa é uma preocupação bastante presente no desempenho da função da maior parte dos profissionais da área.

Sobre o impacto no consumo de água que está diretamente ligado a eficiência energética quando pensamos no consumo das máquinas ao processarem sistemas, 62,1% dos pesquisados responderam nunca pensar nessa questão ao desenvolver um produto de *software*.

Na preocupação com desempenho do processamento 76,9% dos pesquisados apontaram ser uma preocupação presente no seu dia a dia, classificando este impacto nas escalas quatro ou cinco, como um pensamento sempre ou frequentemente presente nas suas atividades.

A qualidade de código é uma preocupação da maior parte dos respondentes, onde 92,6% classificaram na escala quatro ou cinco com presença de todas as faixas com mais dois anos de experiência neste percentual.

Sobre os impactos sociais do desenvolvimento de *software* apenas 26,4% indicaram as escalas quatro ou cinco para pergunta sobre pensar neste impacto ao desenvolver um *software*. Ainda no resultado geral 31,6% dos respondentes nunca pensa em impacto social no desempenho da sua função.

A manutenção de código mostrou-se uma preocupação presente no dia a dia do profissional da área, já que todas as faixas estão presentes nas escalas quatro e cinco com 87,4% das respostas.

A usabilidade foi um dos poucos temas apontados em que nenhum respondente afirmou nunca pensar sobre, as escalas um e dois tiveram zero respostas. Mais de 80% dos pesquisados classificaram a preocupação com esse impacto nas escalas quatro ou cinco.

A importância dos custos no desenvolvimento de *software* está presente entre a maioria das respondentes, onde 89,5% deles classificaram o assunto nas escalas de três a cinco quando pensam nos impactos do seu desenvolvimento.

A redução de despesas que possui relação direta com os custos apresentou resultado similar onde 90,5% dos pesquisados responderam de três a cinco na escala para a pergunta. Curiosamente mais pessoas disseram pensar com mais frequência em redução de despesas do que em custos, menos pessoas a classificaram redução de despesas na escala como cinco do que no item sobre custos.

A pergunta sobre entregar valor ao cliente juntamente com usabilidade e manutenção de código foram as únicas perguntas com zero respostas nas escalas um e dois. O que mostra

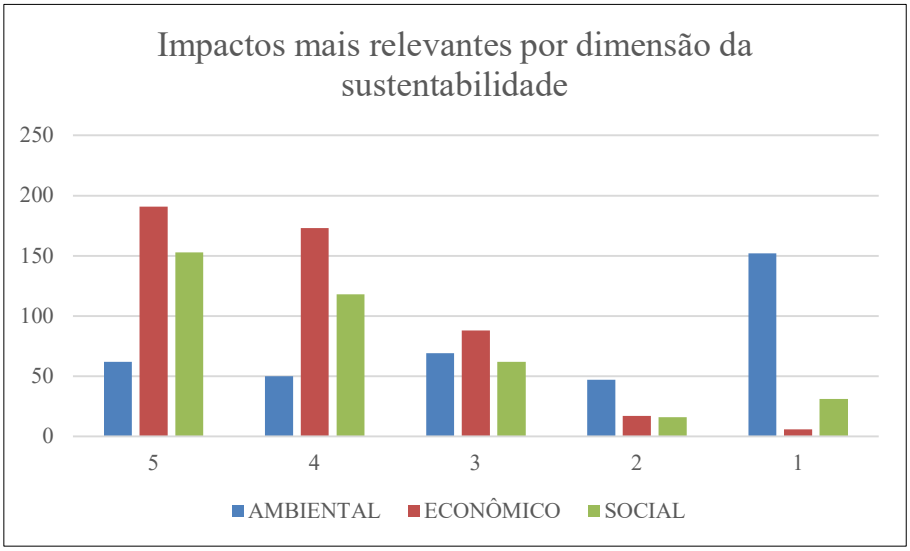
ser um impacto de alta relevância entre os profissionais independente da experiência, e no geral 66,3% classificaram este item como cinco na escala.

A baixa relevância da preocupação com os impactos ambientais durante o ciclo de desenvolvimento de *software* se mostrou bem expressiva com 49,5% dos pesquisados afirmando nunca ter pensado em impactos ambientais no desempenho de suas atividades profissionais.

Quanto à preocupação com a legibilidade de código, 50,5% dos respondentes afirmaram sempre pensar nesse quesito ao desenvolver, este percentual engloba a maioria das repostas de todas as faixas de experiências.

De modo geral a pesquisa mostrou que os respondentes estão mais propensos a enxergar com clareza os impactos diretos do seu trabalho ligados a dimensão econômica da sustentabilidade, do que com os impactos indiretos de suas atividades à dimensão ambiental. Entretanto os impactos ambientais podem gerar impactos econômicos indiretos que não são percebidos pelos respondentes. No gráfico da Figura 10 pode-se observar que os profissionais acreditam que o seu trabalho pode ter bastante relevância nos impactos sociais, e não consideram o mesmo potencial para os impactos ambientais do desenvolvimento de *software*.

Figura 10 - Impactos relevantes por dimensão da sustentabilidade



Fonte: Dados da pesquisa (elaborado própria autora).

3.2.2 Considerações da Pesquisa

Como resultado da pesquisa aplicada pode-se concluir que a preocupação com padrões de desenvolvimento que favoreçam a manutenção, reutilização e legibilidade de código tem alta relevância entre os profissionais de todas as faixas de tempo de experiência. E os impactos classificados como mais presentes nas decisões tomadas pelos profissionais estão relacionados a usabilidade e entrega de valor. Todos esses fatores contribuem para um *software* sustentável, porém ao serem questionados explicitamente sobre as características de impactos ambientais, os respondentes afirmaram não ter conhecimento do assunto ou não o considera relevante no ciclo de desenvolvimento de *software*.

Dado o resultado desta pesquisa, o presente estudo se justifica em relevância e importância, já que ele visa disseminar o conhecimento sobre *software* sustentável entre os profissionais da área ao vincular os padrões conhecidos por eles aos seus respectivos impactos na sustentabilidade.

3.3 Análise Comparativa

Foram comparados os padrões de desenvolvimento mais utilizados/conhecidos dos profissionais de acordo com os resultados da *survey*, com os comportamentos e resultados esperados de um *software* sustentável. Os comportamentos e resultados esperados de um *software* sustentável estão presentes em todo o seu ciclo de vida e com potencial de ação entre os envolvidos, ou seja, desde os usuários na forma como o utilizam até o proprietário da solução em suas tomadas de decisões (AHMAD; HUSSAIN; BAHAROM, 2018; JNR; MAJID, 2017; MORAGA et al., 2017).

Entretanto como na pesquisa de Calero, Moraga e Bertoa (2013), neste estudo faremos um recorte nas características de um *software* sustentável relacionadas a qualidade do *software* presentes na ISO/IEC 25010 que estão relacionadas à forma como ele é desenvolvido, modificado e os seus impactos na eficiência energética, otimização de recursos e a sua perdurabilidade.

Nesse contexto seguimos com três categorias principais que agrupam as características do *software* sustentável que avaliaremos nos padrões de desenvolvimento estudados, que são:

- **Confiabilidade**: dentro dessa categoria são avaliadas as características de maturidade, disponibilidade, tolerância a falhas e capacidade de recuperação.
- **Capacidade de manutenção**: as características que a categoria de manutenção são modularidade, reusabilidade, capacidade de análise, capacidade de modificação e capacidade de ser testável.
- **Portabilidade**: dentre as características de portabilidade estão as capacidades de se adaptar, instalar e ser substituído.

As tecnologias, padrões ou boas práticas utilizados na análise comparativa foram extraídos entre os mais notórios, de acordo com o resultado da pesquisa, somando os percentuais de respondentes nas escalas quatro e cinco para a pergunta que buscou saber o nível de conhecimento dos entrevistados em relação a tecnologia conforme mostra a tabela 3.

Tabela 3 - Padrões ou tecnologias mais comuns

Padrão/Tecnologia	Surgimento	Escala 5	Escala 4	Total
POO - Programação Orientada a Objetos	Século XX	47% (45)	33% (31)	80% (76)
Desenvolvimento de Software Ágil		47% (45)	22% (21)	69% (66)
MVC - Multicamadas		37% (35)	25% (24)	62% (59)
Rest / Restfull		29% (28)	23% (22)	53% (50)
Clean Code		26% (25)	25% (24)	52% (49)
SOA - Arquitetura orientada serviços		26% (25)	24% (23)	51% (48)
Design Patterns		20% (19)	26% (25)	46% (44)
SOLID: Os 5 princípios da POO	Século XXI	16% (15)	23% (22)	39% (37)
GIT Flow		17% (16)	19% (18)	36% (34)
Micro Serviços		21% (20)	14% (13)	35% (33)
Contêineres		19% (18)	13% (12)	32% (30)
DDD – Domain-Driven Design		17% (16)	13% (12)	29% (28)
Software Sustentável		15% (14)	7% (7)	22% (21)

Fonte: Dados da pesquisa (elaborado pela própria autora).

Com o levantamento apresentado na tabela 3, observou-se que os padrões ou tecnologias consideradas mais adotadas são também os mais antigos, com surgimento na segunda metade do século XX, fato esse muito provavelmente vinculado ao perfil dos profissionais pesquisados que em sua maioria são pessoas com mais de dez anos de experiência em desenvolvimento de *software*. Os padrões criados no século XXI como o SOLID por exemplo, se mostraram mais praticados/conhecidos entre profissionais com menos de dois anos de experiência.

Para o quadro comparativo da tabela 4 consideramos os padrões mais recentes, desenvolvidos no século XXI.

Tabela 4 - Análise Comparativa: Padrões x *Software* Sustentável

	Característica do *Software* Sustentável	**SOLID**	**Micro-serviços**	**DDD**	**Git Flow**	**Contêiner**
Confiabilidade	Maturidade		A arquitetura em micro-serviços permite a escalabilidade das aplicações de sistemas e garante a sua maturidade.	Como o DDD prega que o sistema seja orientado ao negócio e que especialistas das duas áreas estejam muito próximos com uma linguagem única, isso implica em menor número de falhas decorrentes de problemas de comunicação e mais maturidade do alinhamento entre sistemas e negócios.	O Git Flow é um processo de controle de versão maduro e testado pela comunidade de profissionais.	
	Disponibilidade		Micro-serviços pregam por resiliência e esse quesito aumenta a disponibilidade de uma aplicação.			A capacidade de um contêiner se autorrecuperar garante alta disponibilidade.
	Tolerância a falhas	Com a utilização dos princípios do SOLID pretende-se desenvolver um código mais limpo e consequentemente menos propenso a falhas.	A flexibilidade e alta escalabilidade com que uma arquitetura de micro-serviços pode ser conduzida permite que mesmo que um serviço pare de funcionar o usuário final não seja impactado por completo ou tenha um impacto apenas parcial na sua usabilidade.	Defende menor número de falhas decorrente da proximidade entre negócios e sistemas.		

	Capacidade de recuperação		Escalabilidade, flexibilidade e resiliência são características e vantagens dos micro-serviços que contribuem para simplificar a capacidade de recuperação de um *software* e torna até mesmo transparente ao usuário final.		A principal função de um sistema de controle de versão é manter o histórico de alterações de um código fonte e possibilitar a recuperação de funções especificas. Com o processo do Git Flow é possível recuperar versões mais segmentadas por funcionalidade e ambiente.	Os contêineres podem ser agrupados e controlados para se recuperarem automaticamente a cada indisponibilidade.
Capacidade de Manutenção	Modularidade	Os princípios SRP e ISP abordam a modularidade do sistema pela segregação de interface que representam contratos do sistema com uma única responsabilidade.	O escopo reduzido dos micro-serviços faz com a modularidade seja um pré-requisito desta arquitetura.	O DDD contribui para orientar na granularidade das aplicações separando por definições de negócios.		A premissa da conteinerização é a modularização geralmente em micro-serviços.
	Reusabilidade	Os cinco princípios do SOLID se baseiam em padrões que favorecem a reusabilidade de código.	O desacoplamento de funcionalidades e negócios permite o reuso dos micro-serviços.			Os contêineres podem ser reutilizados com facilidade.
	Capacidade de análise	O primeiro princípio do SOLID ou SRP prega que cada parte do *software* deve ter uma única responsabilidade, essa característica facilita a análise do *software*.	Um escopo menor de *software* pode ser analisado com mais facilidade.	A modularização do sistema orientado ao negócio contribui na capacidade de análise.		
	Capacidade de modificação	O princípio OCP defende que as classes de um sistema devem estar fechadas para modificação e abertas para extensão. O seu benefício na capacidade de modificação de um *software* pode parecer contraintuitivo pela sua descrição, mas ele contribui para que as modificações do sistema sejam feitas de forma mais segura com	Alterações são feitas com mais facilidade e menos complexidade já o escopo é menor.		O Git flow apresenta um processo seguro de modificação de código fonte por vários profissionais simultaneamente.	

		a extensão de comportamentos em novas implementações, garantindo a facilidade de reverter a modificação já que a versão anterior não foi alterada.				
	Capacidade de ser testável	O princípio ISP juntamente com o princípio de Liskov facilitam a integração com testes de *software* porque trabalham com abstração de classes, com parâmetros de entrada e saída de funções estáveis.	A menor complexidade do serviço facilita a sua capacidade de ser testado.	Possui padrões e boas práticas para os processos da fase de testes.		A utilização de contêineres elimina problemas de diferenças de configuração de máquina, já que uma vez instalado o contêiner se comportara da mesma forma em todas as máquinas.
Portabilidade	Capacidade de adaptação		O uso do consolidado protocolo de comunicação HTTP facilita a adaptação, instalação e substituição dos micro-serviços.			Um dos principais benefícios da conteinerização é a portabilidade e adaptabilidade dos contêineres. Um contêiner pode ser instalado e substituído com poucas instruções de comando.
	Capacidade de instalação					
	Capacidade de substituição	O princípio DIP junto com os princípios responsabilidade única e segregação de interface quando empregados facilitam a substituição de *softwares* integrados ao sistema porque criam uma camada de abstração que pode ser implementado por outro *software* sem afetar o funcionamento do seu consumidor, ou seja, o consumidor não precisa conhecer o sistema que fornece a funcionalidade.				

Fonte: elaborado pela autora.

O quadro mostra que os padrões pesquisados possuem características e resultados que se assemelham ou atingem os requisitos de qualidade esperados de um *software* sustentável. E estes padrões podem ser utilizados de forma conjunta. Existem estudos que sugerem o DDD como o padrão ideal para modularizar a arquitetura de um sistema em micro-serviços (MERSON; YODER, 2020; RADEMACHER; SORGALLA; SACHWEH, 2018), já que a orientação do sistema aos módulos do negócio idealizado no DDD é uma forma de dividir os sistemas por módulos e atingir a granularidade ideal de um micro-serviço, para que este não seja demasiadamente pequeno ou grande para o seu conceito.

Os princípios do SOLID podem ser aplicados a qualquer código de *software* escrito em uma linguagem de programação orientada a objetos, sendo o código pertencente a um micro-serviço ou não, o código estando orientado ao negócio ou não. Ou seja, a junção dos padrões SOLID para o desenvolvimento do código, micro-serviço para a arquitetura do *software* e DDD para o projeto e modularização ou orientação do sistema é perfeitamente factível, dado que um padrão não exclui o outro. Ao mesmo tempo, a tecnologia de contêiner é muito indicada para arquitetura em micro-serviços e o *Git Flow* adequado para o controle de versão de qualquer tipo de código fonte, inclusive de arquivos de configuração de contêineres (SCHERMANN; ZUMBERI; CITO, 2018).

4 CONCLUSÃO

Como defendido por Noureddine e Rajan (2015) e Prechelt et al. (2002) os padrões de projetos promovem resultados como a legibilidade de código, e por consequência facilitam a compreensão do sistema como um todo por parte dos profissionais envolvidos, reduzindo a curva de aprendizado sobre o funcionamento e desenvolvimento deste, colaborando para a manutenção e evolução do código.

Os padrões de projetos atuam principalmente na resolução de problemas comuns na área de sistemas com soluções elegantes, testadas e validadas no mercado por outros profissionais. Todos esses fatores de relevante importância no uso de padrões de desenvolvimento mostram-se reconhecidos pelos profissionais da área, de acordo com a pesquisa realizada durante o presente estudo.

Ao mesmo tempo, o uso destes padrões pode gerar como resultado impactos positivos na sustentabilidade do *software*, visto que melhoram diversos aspectos de qualidade presentes na ISO/IEC 25010. Entretanto na pesquisa realizada os entrevistados mostraram não ter consciência desses efeitos. Por outro lado, na análise comparativa pode-se observar a presença desses aspectos nos padrões e princípios pesquisados, contrapostos às definições de qualidade esperadas de um *software* sustentável como sugerido por Calero, Moraga e Bertoa (2013).

Tanto Calero, Moraga e Bertoa (2013) quanto Gárcia-Mireles et al. (2018) têm o padrão ISO/IEC 25010 como normativa mais adequada para fundamentar as definições e métricas de um *software* sustentável. Todavia, como apresentado no referencial teórico, existem outras normativas designadas ao processo de desenvolvimento de sistemas com potencial para serem aliadas no objetivo de construir sistemas de forma mais sustentável.

O conceito de *software* sustentável defenido por Dick, Naumann e Kuhn (2010) cujos impactos negativos às três dimensões da sustentabilidade ambiental, economica e social (ELKINGTON, 2011) são mínimos ou inexistentes e alguns casos até mesmo positivos, foram identificados no quadro de análise compativa. Contudo a análise comparativa dos padrões de desenvolvimento em relação aos aspectos presentes nas intersecções entre qualidade e sustentabilidade apresentados por García-Mireles et al. (2018), mostram que com o uso correto de padrões conhecidos é possível desenvolver um sistema sustentável, sem necessariamente utilizar um padrão ou guia de boas práticas específico para essa finalidade.

Os benefícios da metodolia ágil no desenvolvimento observados por Martin (2014) e Vallon et al. (2018) se adequam aos aspectos de qualidade da ISO/IEC 25010 esperados de um *software* sustentável relacionados ao nível de confiabilidade deste. Estes achados foram evidenciados no quadro de análise comparativa, destacando-se a interação entre times técnicos e clientes, a melhora de desempenho, e as boas práticas no processo de comunicação que contribuem para aumentar o nível de maturidade do produto.

O *Clean Code*, padrão de desenvolvimento baseado na metodologia ágil, preconizado por Martin (2011) está presente nas características relacionadas a capacidade de manutenção de um sistema, em especial quando o código for bem escrito, simples e objetivo, levando ao aumento da legibilidade e redução de sua complexidade. Estes fatores contribuem para a capacidade de análise e modificação de um sistema. O conceito de responsabilidade única e ausência de duplicações de código garantem a modularidade esperada de um *software* com boa capacidade de manutenção.

O SOLID é um conjunto de princípios originários do *Clean Code* e por consequência herda boa parte das vantages e características relacionadas a qualidade de *software* quanto a capacidade de manutenção. Estes princípios se aplicam a linguagem de programação orientada a objetos, paradigma de alto nível com sintaxe mais próxima aos comportamentos humanos e seus quatro pilares : abstração, herança, encapsulamento e polimorfismo (DEITEL, 2013). Além dos aspectos positivos relacionados a manutenção e tolerância a falhas, os princípios de inversão de depência, segregação de interfaces e responsabilidade única definidos por Martin (2014), quando utilizados em conjunto, proporcionam facilidade na capacidade de subsituição de uma aplicação, aumentando sua portabilidade como evidenciado na análise comparativa mostrada neste estudo.

O padrão de projeto DDD definido por Evans (2010) como um conjunto de boas práticas e padrões de modelagem com objetivo de orientar o ciclo de vida de uma aplicação aos conceitos do negócio, promove uma linguagem única entre times técnicos e clientes a fim de minimizar erros e construir processos assertivos. Essas características foram observadas no quadro comparativo como influenciadores na confiabilidade e capacidade de manutenção de um sistema, por garantir o alinhamento dos envolvidos em todas as etapas do ciclo de vida do produto.

Segundo Merson e Yoder (2020) o DDD é um padrão de projeto aliado a construção de aplicações na arquitetura em micro-serviços, já que a sua modularização em conceitos do negócio permitem uma granularidade ideal para esta arquitetura, e esta perspectiva está diratamente ligada aos requisitos de qualidade relacionados a capacidade de manutenção de um *software*. Ao considerar que aplicações menores geralmente são menos complexas, e portanto mais resistentes a falhas, estas favorecem as alterações independentes ao não impactarem em outros sistemas com os quais estão relacionados, como sugerido por Pautasso et al. (2017).

A tecnologia de contêineres como exposto por Shermann, Zumberi e Cito (2018) é muito utilizada em arquiteturas de micro-serviços e esta junção provê alto nível de portabilidade ao sistema, aspecto este presente nas características de qualidade esperadas de um *software* sustentável além de aumentar a disponibilidade como observado durante análise.

Já a adoção do *Git Flow* como padrão de repositórios para versionamento de código fonte e arquivos de configuração de um sistema (German, Adams e Hassan, 2015) apresenta boas práticas e diretrizes que permitem a recuperação do estado de uma aplicação por seu histórico de alterações, e com seu sistema de ramificações possibilita o trabalho simultâneo de grandes de equipes de desenvolvimento no mesmo sistema.

Deste modo considerando as caracteristicas da tecnologia de conteineres e do *Git Flow* estes se enquadram no aspecto de confiabilidade do *software*, por aumentar a maturidade e capacidade de recuperação e de manutenção, viabilizando que as modificações sejam realizadas de maneira segura e com possibilidade de serem desfeitas com facilidade.

A pesquisa eletronica (*survey*) realizada para este estudo, possibilitou evidenciar que os padrões e boas práticas escolhidos para análise e elaboração de códigos/programas (micro-serviços, SOLID, DDD, *Git Flow* e Contêineres) são muito utilizadas pelos profissionais de desenvolvimento. Ao mesmo tempo permitiu identificar que conceito de *software* sustentável não é comum entre os profissionais, confirmando a lacuna de pesquisa abordada nesta monografia.

Tomando como base os resultados obtidos com a survey realizada para este estudo, sugere-se que os padrões e paradigmas que favorecem as práticas sustentáveis como reusabilidade de código, alta disponibilidade, capacidade de recuperação, portabilidade, menor custo com manutenção decorrente de menor número de erros, e maior número de testes na programação são aceitos e amplamente utilizados pelos profissionais entrevistados, o que os transforma em

mecanismos eficientes para disseminação dos conceitos e práticas de desenvolvimneto de *softwares* sustentáveis, bem como pavimenta um caminho seguro na implementacao de tais práticas.

Ademais ao evidenciar os impactos positivos às dimensões da sustentabilidade quando da utilização dos padõres micro-serviços, SOLID, DDD, *Git Flow* e Contêineres adiciona-se mais um fator de motivação para a adoção de boas práticas e padronização de projetos e desenvolvimento de *software* tanto pelas empresas quanto pelos profissionais da área.

Entretanto como o uso de padrões e boas práticas não são obrigátorios, sendo totalmente possível construir um sistema que funcione, e que atenda os requisitos funcionais e de usabilidade sem qualquer padrão conhecido.Todavia conforme evidenciado no quadro comparativo, os benefícios que a adoção dos padrões agrega aos sistemas e aos profissionais, dificilmente serão alcançados em sua ausência, resssaltando-se a necessidade de promover o uso dos referidos padrões.

Diante das análises realizadas e com os dados apurados por meio da survey, conclui-se que a sustentabilidade apesar de encontrar-se na agenda dos empresários, governos e sociedade civil ainda não encontra nas atividades de desenvolvimento de *software* a adesão desejada, quer seja por ausencia de conhecimento, falhas na divulgação da relação entre os padrões e as práticas sustentáveis, ou ainda pelo distanciamento ou independencia que estes profissionais possuem ao executar suas tarefas diárias.

O presente estudo além de identificar os benefícios e importância dessas práticas, reforça a sua utilização para uma finalidade que não finda no ciclo de vida *software* mas se estende às três dimensões da sustentabilidade - social, economica e ambiental.

4.1 Contribuições do trabalho

O presente estudo contribui com a identificação dos apectos e características sustentáveis presentes nas boas práticas dos princípios do SOLID, no padrão de projeto DDD, no padrão de arquitetura em Micro-serviços, Contêineres e no fluxo de trabalho apoiado pelo *Git Flow*. Estas evidências viabilizam a compreensão do *software* sustentável por parte de profissionais de desenvolvimento que já fazem uso desses padrões ou boas práticas. E por fim confirma ser possível desenvolver *softwares* sustentáveis sem a necessidade de criar um framework ou guia

específico para este fim, o que provavelmente necessitaria de uma curva de aprendizado para ser adotado.

O estudo também evidenciou (tabela 3) que a adoção e difusão de um padrão pode ser lento, pela falta da constante atualiazação de conhecimentos entre as gerações de profissionais, fator este que aumenta ainda mais a curva de aceitação e utilização de um novo padrão.

4.2 Trabalhos futuros

Como trabalho futuro é proposto a análise comparativa de padrões de desenvolvimento no paradigma da programação estruturada, já que o presente trabalho avaliou padrões voltados ao paradigma da programação orientada a objetos.

Propoem-se ainda a realização de estudo acerca dos impactos do desenvolvimento de *software* à dimensão social da sustentabilidade, visto que durante a pesquisa os profissionais participantes demonstraram maior preocupação com os impactos sociais do seu trabalho do que com os impactos ambientais.

Outro ponto não explorado neste trabalho, mas que requer uma análise aprofundada, recai sobre a existência ou não de padrões ou boas práticas com impactos negativos à sustentabilidade, e que devam ser evitados ou mitigados com ações de contorno ou mesmo com o emprego de novas tecnologias e/ou processos.

REFERÊNCIAS BIBLIOGRÁFICA

AHMAD, Ruzita; HUSSAIN, Azham; BAHAROM, Fauziah. Integrated-Software Sustainability Evaluation Model (i-SSEM) development. **Journal of Telecommunication, Electronic and Computer Engineering**, *[S. l.]*, v. 10, n. 1–11, p. 39–46, 2018.

ARCELLI, Davide; DI POMPEO, Daniele. Applying Design Patterns to Remove Software Performance Antipatterns: A Preliminary Approach. **Procedia Computer Science**, *[S. l.]*, v. 109, n. 2016, p. 521–528, 2017. DOI: 10.1016/j.procs.2017.05.330.

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS - ABNT. ABNT NBR ISO 14040 Gestão ambiental - Avaliação do ciclo de vida - Princípios e estrutura. **Associação Brasileira de Normas Técnicas**, *[S. l.]*, p. 1–22, 2014.

BETZ, Stefanie; CAPORALE, Timm. Sustainable software system engineering. **Proceedings - 4th IEEE International Conference on Big Data and Cloud Computing, BDCloud 2014 with the 7th IEEE International Conference on Social Computing and Networking, SocialCom 2014 and the 4th International Conference on Sustainable Computing and C**, *[S. l.]*, p. 612–619, 2015. DOI: 10.1109/BDCloud.2014.113.

BHARATHI, R.; SELVARANI, R. **A framework for the estimation of OO software reliability using design complexity metrics2015 International Conference on Trends in Automation, Communications and Computing Technology (I-TACT-15)**, 2015. DOI: 10.1109/ITACT.2015.7492648.

BOGNER, Justus; FRITZSCH, Jonas; WAGNER, Stefan; ZIMMERMANN, Alfred. Limiting technical debt with maintainability assurance: An industry survey on used techniques and differences with service- and microservice-based systems. **Proceedings - International Conference on Software Engineering**, *[S. l.]*, p. 125–133, 2018. DOI: 10.1145/3194164.3194166.

BOWLES, John B. Code from requirements: New productivity tools improve the reliability and maintainability of software systems. **Proceedings of the Annual Reliability and Maintainability Symposium**, *[S. l.]*, p. 68–72, 2004. DOI: 10.1109/rams.2004.1285425.

BRUNDTLAND, G. H. Our common future - Call for action. *In*: ENVIRONMENTAL CONSERVATION 1987, **Anais** [...]. [s.l: s.n.] p. 16.

CALERO, C.; PIATTINI, M. Introduction to Green in Software Engineering. *In*: **Green in Software Engineering**. Londres: Springer, 2015. p. 3–24.

CALERO, Coral; BERTOA, Manuel F.; MORAGA, Ma Ángeles. A systematic literature review for software sustainability measures. **2013 2nd International Workshop on Green and Sustainable Software, GREENS 2013 - Proceedings**, *[S. l.]*, p. 46–53, 2013. DOI: 10.1109/GREENS.2013.6606421.

CALERO, Coral; MORAGA, M. Angeles; BERTOA, Manuel F. Towards a Software Product Sustainability Model. *[S. l.]*, v. 25010, 2013.

CHANG, Che Yu; OU, Pei Pei; DENG, Der Jiunn. Cross-site large-scale software delivery with enhanced git branch model. **Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS**, *[S. l.]*, v. 2019-Octob, n. 1, p. 153–156, 2019. DOI: 10.1109/ICSESS47205.2019.9040834.

CHANG, Danni; LEE, C. K. M.; CHEN, Chun Hsien. Review of life cycle assessment towards sustainable product development. **Journal of Cleaner Production**, *[S. l.]*, v. 83, p. 48–60, 2014. DOI: 10.1016/j.jclepro.2014.07.050.

CITO, Jürgen; GALL, Harald C. Using docker containers to improve reproducibility in software engineering research. **Proceedings - International Conference on Software Engineering**, *[S. l.]*, p. 906–907, 2016. DOI: 10.1145/2889160.2891057.

COCKBURN, A.; HIGHSMITH, J. Agile software development, the people factor. **Computer**, *[S. l.]*, v. 34, n. 11, p. 131–133, 2001. DOI: 10.1109/2.963450. Disponível em: http://ieeexplore.ieee.org/document/963450/. Acesso em: 18 mar. 2018.

DEITEL, Paul; DEITEL, Harvey. **Java: Como Programar**. Pearson educacion, 2008, DOI: 10.1017/CBO9781107415324.004.

DICK, Markus; NAUMANN, Stefan; KUHN, Norbert. A model and selected instances of green and sustainable software. **IFIP Advances in Information and Communication Technology**, *[S. l.]*, v. 328, p. 248–259, 2010. DOI: 10.1007/978-3-642-15479-9_24.

DOCKER. **What is a Container?** [s.d.].

DUTIL, Daniel et al. Software quality engineering in the new ISO standard: ISO/IEC 24748-systems and software engineering---guide for life cycle management. *In*: PROCEEDINGS OF

THE THIRD C* CONFERENCE ON COMPUTER SCIENCE AND SOFTWARE ENGINEERING 2010, **Anais** [...]. [s.l: s.n.] p. 89–96.

ELKINGTON, J. **Sustentabilidade. Canibais com Garfo e Faca**. São Paulo : Makron Books, 2001.

ERIC FREEMAN, ELISABETH FREEMAN, BERT BATES, Kathy Sierra. **Head First Design Patterns**. [s.l: s.n.]. v. 34 DOI: 10.1093/carcin/bgt051.

EVANS, Eric. Domain-Driven Design ATA C A N D O A S C O M P L E X I D A D E S. *[S. l.]*, 2010.

FEATHERS, Michael C. This Class Is Too Big And I Don't Want It to Get Any Bigger. *In*: **Working Effectively With Legacy Code**. [s.l.] : Pearson, 2004. p. 245–321.

FERNANDES, Filipe; WERNER, Claudia. Towards immersive learning in object-oriented paradigm: A preliminary study. **Proceedings - 2019 21st Symposium on Virtual and Augmented Reality, SVR 2019**, *[S. l.]*, p. 59–68, 2019. DOI: 10.1109/SVR.2019.00026.

FOWLER, Martin. **Is Design Dead?** 2004. Disponível em: https://www.martinfowler.com/articles/designDead.html. Acesso em: 18 mar. 2018.

GAO, Kai Zhou; PAN, Quan Ke; SUN, Qiang Qiang. Conceptual design knowledge retrieval, reuse and innovation based on case and constraint. **Proceedings - 2009 2nd International Workshop on Knowledge Discovery and Data Mining, WKKD 2009**, *[S. l.]*, p. 163–166, 2009. DOI: 10.1109/WKDD.2009.31.

GARCÍA-MIRELES, Gabriel Alberto; MORAGA, Mª Ángeles; GARCÍA, Félix; CALERO, Coral; PIATTINI, Mario. Interactions between environmental sustainability goals and software product quality: A mapping study. **Information and Software Technology**, *[S. l.]*, v. 95, n. October 2017, p. 108–129, 2018. DOI: 10.1016/j.infsof.2017.10.002. Disponível em: https://doi.org/10.1016/j.infsof.2017.10.002.

GARCÍA-RODRÍGUEZ DE GUZMÁN, Ignacio; PIATTINI, Mario; PÉREZ-CASTILLO, Ricardo. **Green software maintenance**. [s.l: s.n.]. DOI: 10.1007/978-3-319-08581-4_9.

GERMAN, Daniel M.; ADAMS, Bram; HASSAN, Ahmed E. A dataset of the activity of the git super-repository of Linux in 2012. **IEEE International Working Conference on Mining Software Repositories**, *[S. l.]*, v. 2015- Augus, p. 470–473, 2015. DOI:

10.1109/MSR.2015.66.

GILL, Asif Qumer; HENDERSON-SELLERS, Brian; NIAZI, Mahmood. Scaling for agility: A reference model for hybrid traditional-agile software development methodologies. **Information Systems Frontiers**, *[S. l.]*, v. 20, n. 2, p. 315–341, 2018. DOI: 10.1007/s10796-016-9672-8.

GONÇALVES-DIAS, Sylmara Lopes Francelino. Sustentabilidade. *In*: **Dicionário para a formação em gestão social**. Salvador. p. 165–168.

GREEN, Graham; KENNEDY, Paul; MCGOWN, Alistair. Management of multi-method engineering design research: a case study. **Journal of Engineering and Technology Management**, *[S. l.]*, v. 19, n. 2, p. 131–140, 2002. DOI: 10.1016/S0923-4748(02)00006-1. Disponível em: http://www.sciencedirect.com/science/article/pii/S0923474802000061.

GUPTA, Rajeev Kumar; MANIKREDDY, Prabhulinga; NAIK, Sandesh; ARYA, K. Pragmatic approach for managing technical debt in legacy software project. **ACM International Conference Proceeding Series**, *[S. l.]*, v. 18-20- Febr, p. 170–176, 2016. DOI: 10.1145/2856636.2856655.

HAZZAN, Orit; DUBINSKY, Yael. The Agile Manifesto. **SpringerBriefs in Computer Science**, *[S. l.]*, n. 9783319101569, p. 9–14, 2014. DOI: 10.1007/978-3-319-10157-6_3.

ISO/IEC 25010. **Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models**, 2011.

JAAFAR, Fehmi; LOZANO, Angela; GUÉHÉNEUC, Yann Gaël; MENS, Kim. Analyzing software evolution and quality by extracting Asynchrony change patterns. **Journal of Systems and Software**, *[S. l.]*, v. 131, p. 311–322, 2017. DOI: 10.1016/j.jss.2017.05.047.

JNR, Bokolo Anthony; MAJID, Mazlina Abdul. Software Based Organizations. *[S. l.]*, p. 26–35, 2017.

JONES, Alex K.; LIAO, Liang; COLLINGE, William O.; XU, Haifeng; SCHAEFER, Laura A.; LANDIS, Amy E.; BILEC, Melissa M. Green computing: A life cycle perspective. **2013 International Green Computing Conference Proceedings, IGCC 2013**, *[S. l.]*, p. 1–6, 2013. DOI: 10.1109/IGCC.2013.6604497.

KATZ, Daniel S. et al. Community organizations: Changing the culture in which research

software is developed and sustained. **Computing in Science and Engineering**, *[S. l.]*, v. 21, n. 2, p. 8–24, 2019. DOI: 10.1109/MCSE.2018.2883051.

KERN, Eva; HILTY, Lorenz M.; GULDNER, Achim; MAKSIMOV, Yuliyan V.; FILLER, Andreas; GRÖGER, Jens; NAUMANN, Stefan. Sustainable software products—Towards assessment criteria for resource and energy efficiency. **Future Generation Computer Systems**, *[S. l.]*, v. 86, p. 199–210, 2018. DOI: 10.1016/J.FUTURE.2018.02.044.

LAMAS, Wendell de Queiroz; GIACAGLIA, Giorgio Eugenio Oscare. The Brazilian energy matrix: Evolution analysis and its impact on farming. **Energy Policy**, *[S. l.]*, v. 63, p. 321–327, 2013. DOI: 10.1016/j.enpol.2013.09.009.

LAMI, Giuseppe; FABBRINI, Fabrizio; BUGLIONE, Luigi. An ISO/IEC 33000-compliant measurement framework for software process sustainability assessment. **Proceedings - 2014 Joint Conference of the International Workshop on Software Measurement, IWSM 2014 and the International Conference on Software Process and Product Measurement, Mensura 2014**, *[S. l.]*, p. 50–59, 2014. DOI: 10.1109/IWSM.Mensura.2014.34.

LATTE, Bjorn; HENNING, Soren; WOJCIESZAK, Maik. Clean code: on the use of practices and tools to produce maintainable code for long-living software. **CEUR Workshop Proceedings**, *[S. l.]*, v. 2308, p. 96–99, 2019.

LE, Duc Minh; DANG, Duc Hanh; NGUYEN, Viet Ha. Generative software module development: A domain-driven design perspective. **Proceedings - 2017 9th International Conference on Knowledge and Systems Engineering, KSE 2017**, *[S. l.]*, v. 2017- Janua, p. 77–82, 2017. DOI: 10.1109/KSE.2017.8119438.

LEPMETS, Marion; RAS, Eric; RENAULT, Alain. A quality measurement framework for IT services. **Proceedings - 2011 Annual SRII Global Conference, SRII 2011**, *[S. l.]*, p. 767–774, 2011. DOI: 10.1109/SRII.2011.84.

LINDSTROM, Lowell; JEFFRIES, R. Extreme programming and agile software development methodologies. **Information Systems Management**, *[S. l.]*, v. 21, p. 41–52, 2004. DOI: 10.1201/1078/44432.21.3.20040601/82476.7. Disponível em: http://www.tandfonline.com/doi/pdf/10.1201/1078/44432.21.3.20040601/82476.7.

LINGAYAT, Ashish; BADRE, Ranjana R.; GUPTA, Anil Kumar. Performance Evaluation for Deploying Docker Containers on Baremetal and Virtual Machine. **Proceedings of the 3rd**

International Conference on Communication and Electronics Systems, ICCES 2018, *[S. l.]*, n. Icces, p. 1019–1023, 2018. DOI: 10.1109/CESYS.2018.8723998.

MADHUMATHI, R. The Relevance of Container Monitoring Towards Container Intelligence. **2018 9th International Conference on Computing, Communication and Networking Technologies, ICCCNT 2018**, *[S. l.]*, p. 8–12, 2018. DOI: 10.1109/ICCCNT.2018.8493766.

MARTIN, Robert C. **Código Limpo: Habilidades Práticas do Agile Software**. Rio de Janeiro: Altas Books, 2011.

MARTIN, Robert C. **Agile Software Development: Principles, Patterns, and Practices**. [s.l: s.n.]. v. 53 DOI: 10.1002/pfi.21408.

MERSON, Paulo; YODER, Joseph. Modeling Microservices with DDD. **Proceedings - 2020 IEEE International Conference on Software Architecture Companion, ICSA-C 2020**, *[S. l.]*, p. 7–8, 2020. DOI: 10.1109/ICSA-C50368.2020.00010.

MOLLA, Alemayehu; COOPER, Vanessa A. IT and Eco-sustainability- Developing and Validating a Green IT R.pdf.crdownload. *[S. l.]*, 2009.

MORAGA, Ma Ángeles; GARCÍA-RODRÍGUEZ DE GUZMÁN, Ignacio; CALERO, Coral; JOHANN, Timo; ME, Gianantonio; MÜNZEL, Harald; KINDELSBERGER, Julia. GreCo: Green code of ethics. **Journal of Software: Evolution and Process**, *[S. l.]*, v. 29, n. 2, p. 1–10, 2017. DOI: 10.1002/smr.1850.

MURUGESAN, San; GANGADHARAN, G. R. Harnessing Green It: Principles and Practices. **Harnessing Green It: Principles and Practices**, *[S. l.]*, n. February, 2012. DOI: 10.1002/9781118305393.

NAUMANN, Stefan; DICK, Markus; KERN, Eva; JOHANN, Timo. The GREENSOFT Model: A reference model for green and sustainable software and its engineering. **Sustainable Computing: Informatics and Systems**, *[S. l.]*, v. 1, n. 4, p. 294–304, 2011. DOI: 10.1016/j.suscom.2011.06.004. Disponível em: http://dx.doi.org/10.1016/j.suscom.2011.06.004.

NIDUMOLU, RAM; PRAHALAD, C. K. .. Rangaswami M. R. Sustainability is the key driver of innovation. **71st World Foundry Congress: Advanced Sustainable Foundry, WFC 2014**, *[S. l.]*, n. September, p. 57–64, 2014.

NIERSTRASZ, Oscar. A Survey of Object-Oriented Concepts. **ACM Press and Addison-Wesley**, *[S. l.]*, p. 3–21, 1989.

NOUREDDINE, Adel; RAJAN, Ajitha. Optimising Energy Consumption of Design Patterns. **Proceedings - International Conference on Software Engineering**, *[S. l.]*, v. 2, p. 623–626, 2015. DOI: 10.1109/ICSE.2015.208.

O'BRIEN, Martin; DOIG, Alison; CLIFT, Roland. Social and environmental life cycle assessment (SELCA): Approach and methodological development. **International Journal of Life Cycle Assessment**, *[S. l.]*, v. 1, n. 4, p. 231–237, 1996. DOI: 10.1007/BF02978703.

OKTAFIANI, Intan; HENDRADJAYA, Bayu. Software Metrics Proposal for Conformity Checking of Class Diagram to SOLID Design Principles. **Proceedings of 2018 5th International Conference on Data and Software Engineering, ICoDSE 2018**, *[S. l.]*, 2018. DOI: 10.1109/ICODSE.2018.8705857.

PAUTASSO, Cesare; ZIMMERMANN, Olaf; AMUNDSEN, Mike; LEWIS, James; JOSUTTIS, Nicolai. Microservices in Practice , Part 2. *[S. l.]*, n. April, p. 97–104, 2017.

PRECHELT, Lutz; UNGER-LAMPRECHT, Barbara; PHILIPPSEN, Michael; TICHY, Walter F. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. **IEEE Transactions on Software Engineering**, *[S. l.]*, v. 28, n. 6, p. 595–606, 2002. DOI: 10.1109/TSE.2002.1010061.

PROCACCIANTI, Giuseppe; FERNÁNDEZ, Héctor; LAGO, Patricia. Empirical evaluation of two best practices for energy-efficient software development. **Journal of Systems and Software**, *[S. l.]*, v. 117, p. 185–198, 2016. DOI: 10.1016/J.JSS.2016.02.035.

RADEMACHER, Florian; SORGALLA, Jonas; SACHWEH, Sabine. Challenges of Domain-Driven Microservice Design. **IEEE Software**, *[S. l.]*, p. 36–43, 2018. DOI: 10.1109/MS.2018.2141028.

RODRIGUES, Bruno Rafael de Oliveira; SOUZA, Daniel Edilson De; FIGUEIREDO, Eduardo Magno Lages. Medindo Acoplamento em Software Orientado a Objeto: Uma Perspectiva do Desenvolvedor DOI - 10.5752/P.2316-9451.2014v3n1p3. **Abakós**, *[S. l.]*, v. 3, n. 1, p. 3–17, 2014. DOI: 10.5752/p.2316-9451.2014v3n1p3.

SÁ-SILVA, Jackson Ronie; ALMEIDA, Crstóvão Domingos; GUINDANI, Joel Felipe.

Pesquisa documental: pistas teóricas e metodológicas. **Revista Brasileira de História & Ciências Sociais**, *[S. l.]*, v. 1, n. 1, p. 1–15, 2009.

SAITO, Yusuke; FUJIWARA, Kenji; IGAKI, Hiroshi; YOSHIDA, Norihiro; IIDA, Hajimu. How do GitHub Users Feel with Pull-Based Development? **Proceedings - 7th International Workshop on Empirical Software Engineering in Practice, IWESEP 2016**, *[S. l.]*, p. 7–11, 2016. DOI: 10.1109/IWESEP.2016.19.

SCHERMANN, Gerald; ZUMBERI, Sali; CITO, Jürgen. Structured information on state and evolution of dockerfiles on github. **Proceedings - International Conference on Software Engineering**, *[S. l.]*, p. 26–29, 2018. DOI: 10.1145/3196398.3196456.

SHARMA, Ratnesh; SHAH, Amip; BASH, Cullen; CHRISTIAN, Tom; PATEL, Chandrakant. Water efficiency management in datacenters: Metrics and methodology. **2009 IEEE International Symposium on Sustainable Systems and Technology, ISSST '09 in Cooperation with 2009 IEEE International Symposium on Technology and Society, ISTAS**, *[S. l.]*, p. 1–6, 2009. DOI: 10.1109/ISSST.2009.5156773.

SHEPPERD, M. Early life-cycle metrics and software quality models. **Information and Software Technology**, *[S. l.]*, v. 32, n. 4, p. 311–316, 1990. DOI: 10.1016/0950-5849(90)90065-Y.

SINGH, Raghu. International Standard ISO/IEC 12207 Software Life Cycle Processes. **Software Process: Improvement and Practice**, *[S. l.]*, v. 2, n. 1, p. 35–50, 1996. DOI: 10.1002/(sici)1099-1670(199603)2:1<35::aid-spip29>3.3.co;2-v.

SMITH, Spencer; JEGATHEESAN, Thulasi; KELLY, Diane. Advantages, disadvantages and misunderstandings about document driven design for scientific software. **Proceedings of SE-HPCCSE 2016: 4th International Workshop on Software Engineering or High Performance Computing in Computational Science and Engineering - Held in conjunction with SC 2016: The International Conference for High Performance Computing, Netwo**, *[S. l.]*, p. 41–48, 2017. DOI: 10.1109/SE-HPCCSE.2016.10.

SOARES, Michel dos Santos. Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software. **Revista Eletrônica de Sistemas de Informação**, *[S. l.]*, v. 3, p. 1–8, 2004. DOI: 105329/resi. Disponível em: http://revistas.facecla.com.br/index.php/reinfo/article/viewArticle/146.

SOMMERVILLE, Ian. **Engenharia de Software 9Th**. [s.l: s.n.].

STANDARDIZATION, F. O. R.; NORMALISATION, D. E. **International Standard Iso**. [s.l: s.n.]. v. 1987

SUWANYA, Suphak; KURUTACH, Werasak. An analysis of software process improvement for sustainable development in Thailand. **Proceedings - 2008 IEEE 8th International Conference on Computer and Information Technology, CIT 2008**, *[S. l.]*, n. Iso 9000, p. 724–729, 2008. DOI: 10.1109/CIT.2008.4594764.

TUSJUNT, Mathawee; VATANAWOOD, Wiwat. Refactoring orchestrated web services into microservices using decomposition pattern. **2018 IEEE 4th International Conference on Computer and Communications, ICCC 2018**, *[S. l.]*, p. 609–613, 2018. DOI: 10.1109/CompComm.2018.8781036.

VALLON, Raoul; DA SILVA ESTÁCIO, Bernardo José; PRIKLADNICKI, Rafael; GRECHENIG, Thomas. Systematic literature review on agile practices in global software development. **Information and Software Technology**, *[S. l.]*, v. 96, n. December 2017, p. 161–180, 2018. DOI: 10.1016/j.infsof.2017.12.004.

VEIGA, José Eli Da. **Sustentabilidade: a legitimação de um novo valor**. São Paulo: Editora Senac, 2019.

WAZLAWICK, Raul Sidnei. Metodologia de Pesquisa em Ciência da Computação. *[S. l.]*, p. 184, 2009.

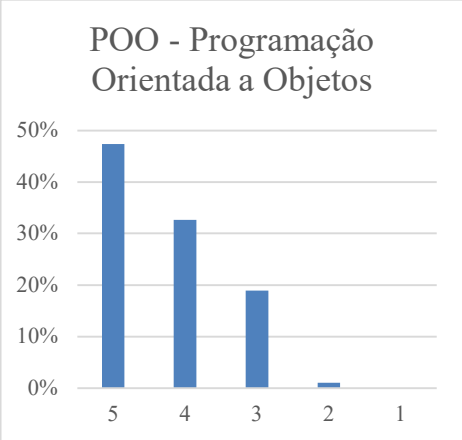
YACOUB, Sherif M.; AMMAR, Hany H. Pattern-oriented analysis and design (POAD): A structural composition approach to glue design patterns. **"Technology of Object-Oriented Languages and Systems**, *[S. l.]*, n. TOOL 34, p. 273–282, 2000. DOI: 10.1109/TOOLS.2000.868978.

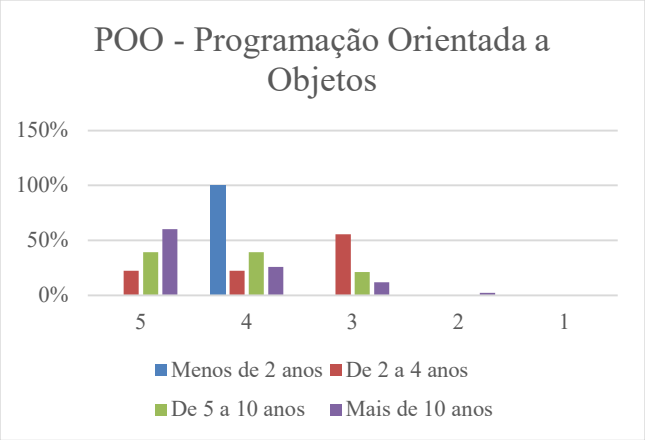
ANEXO A - RESULTADOS DA PESQUISA: DESENVOLVIMENTO DE SOFTWARE SUSTENTÁVEL

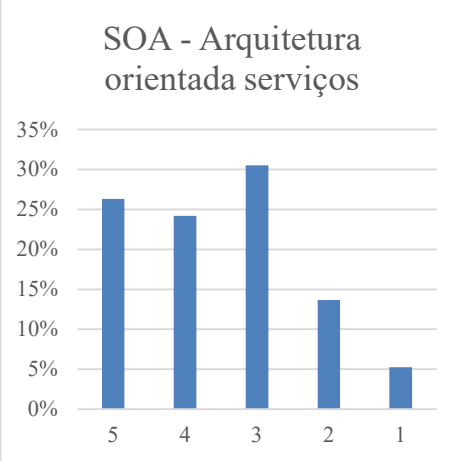
A.1. Resultados da seção 3

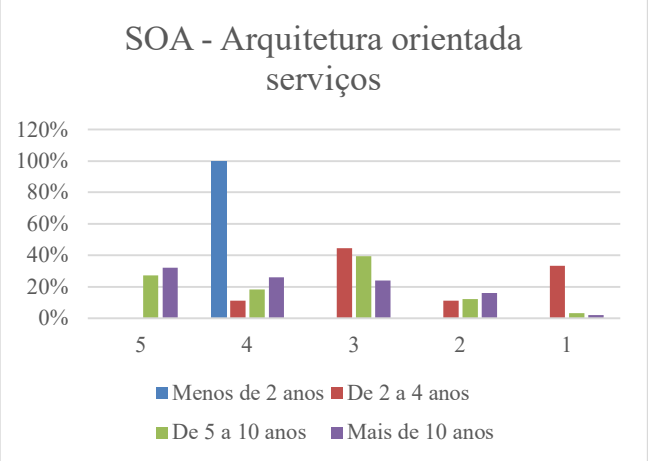
O quanto você avalia que conhece sobre os itens abaixo:

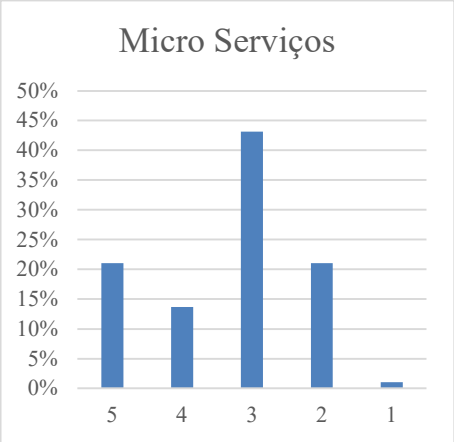
(Em uma escala de 1 a 5, sendo 1 nunca ter ouvido falar sobre e 5 conhecer e utilizar a técnica/padrão no seu dia a dia)

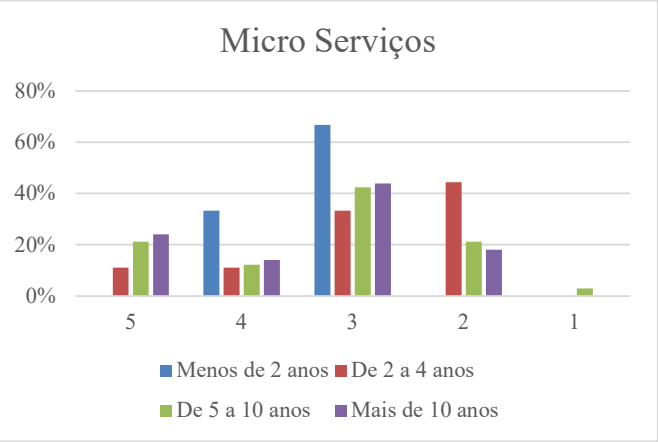




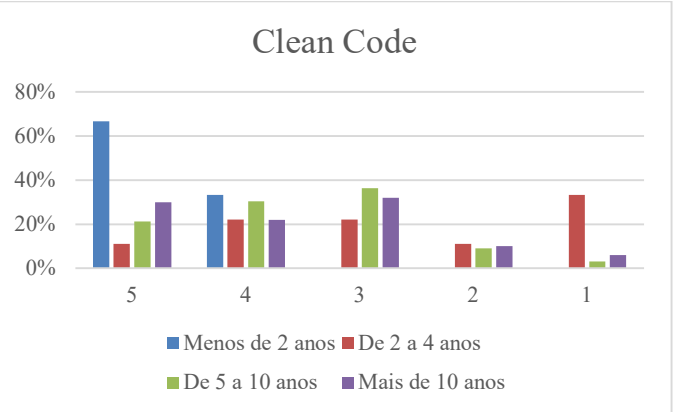


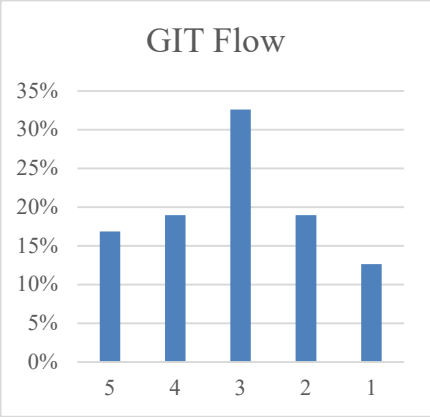


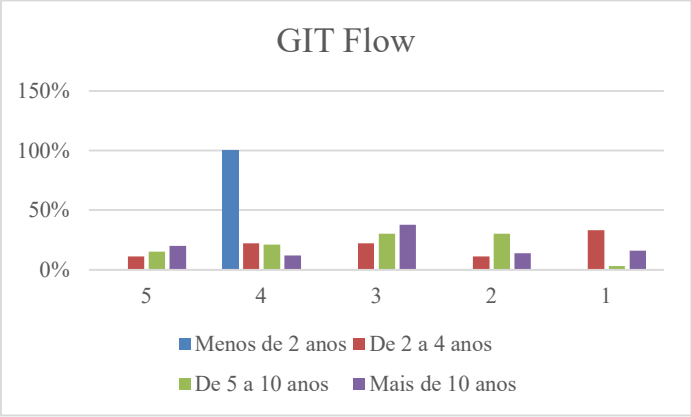

Micro Serviços
50%
45%
40%
35%
30%
25%
20%
15%
10%
5%
0%
5
4
3
2
1

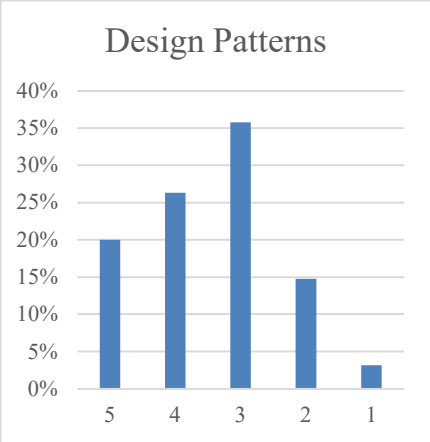

Micro Serviços
80%
60%
40%
20%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

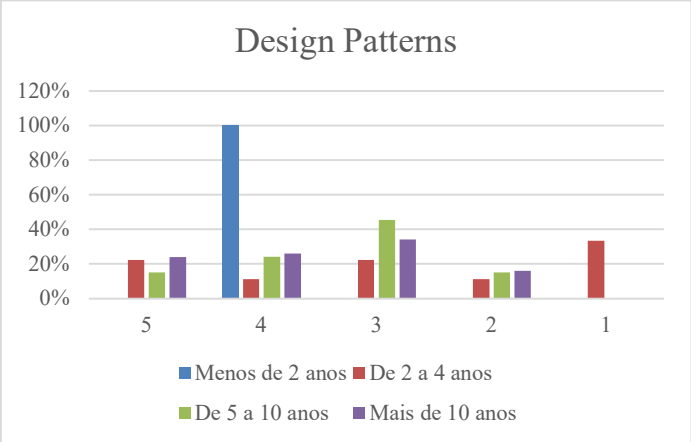

Clean Code
35%
30%
25%
20%
15%
10%
5%
0%
5
4
3
2
1

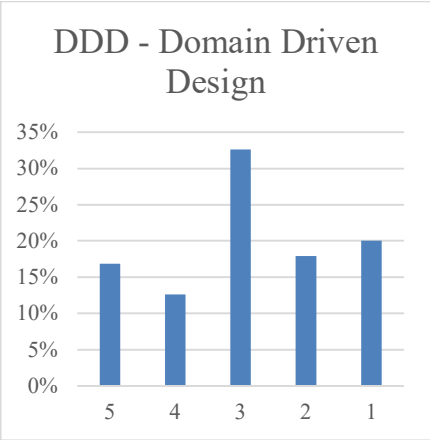

Clean Code
80%
60%
40%
20%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

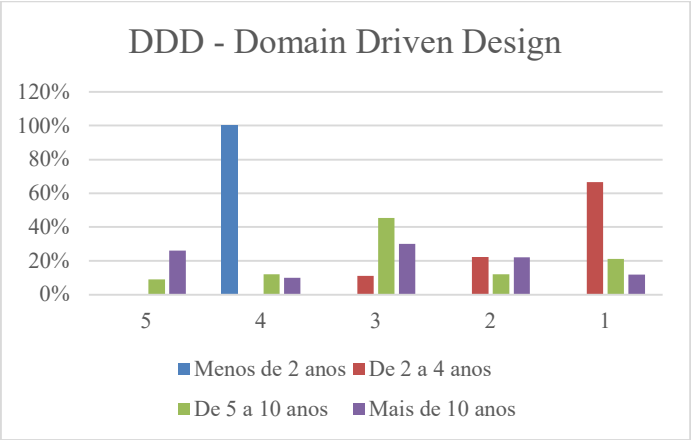

GIT Flow
35%
30%
25%
20%
15%
10%
5%
0%
5
4
3
2
1

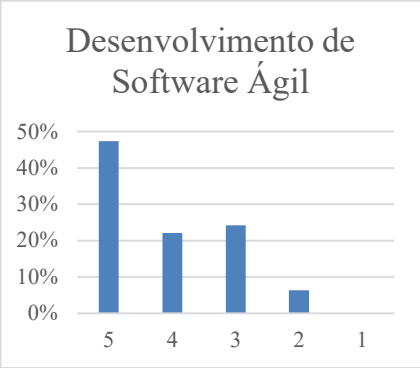

GIT Flow
150%
100%
50%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

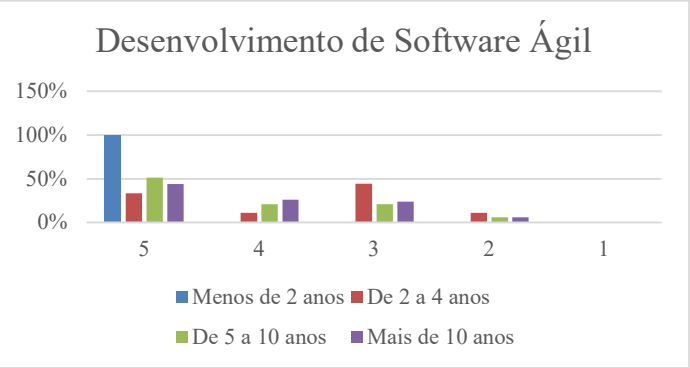

Design Patterns
40%
35%
30%
25%
20%
15%
10%
5%
0%
5
4
3
2
1

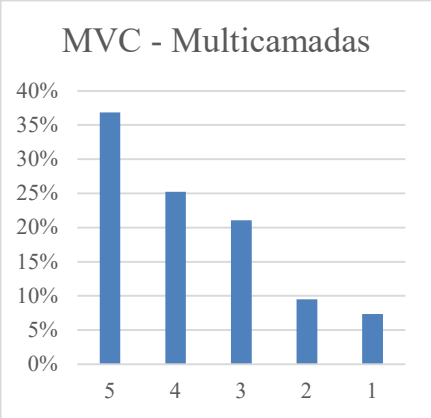

Design Patterns
120%
100%
80%
60%
40%
20%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

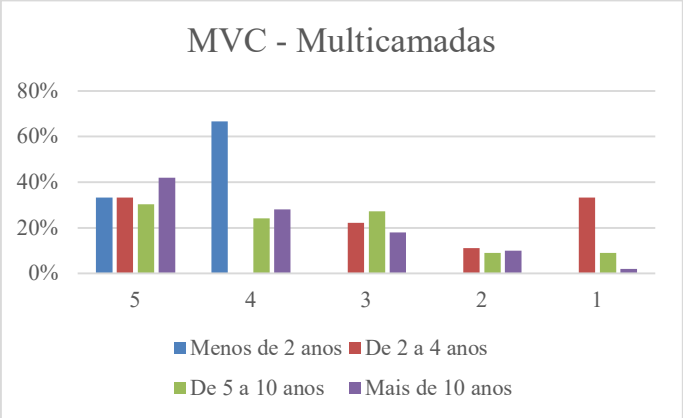

DDD - Domain Driven Design
35%
30%
25%
20%
15%
10%
5%
0%
5
4
3
2
1

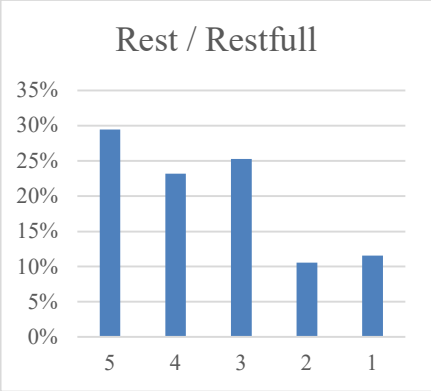

DDD - Domain Driven Design
120%
100%
80%
60%
40%
20%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

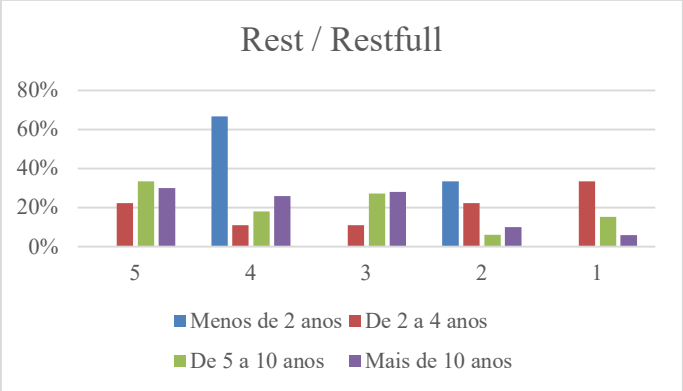

Desenvolvimento de Software Ágil
50%
40%
30%
20%
10%
0%
5
4
3
2
1

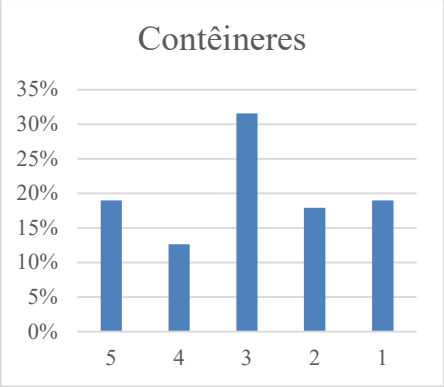

Desenvolvimento de Software Ágil
150%
100%
50%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

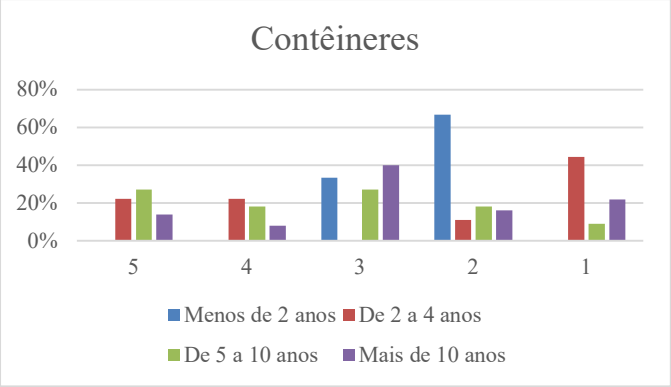








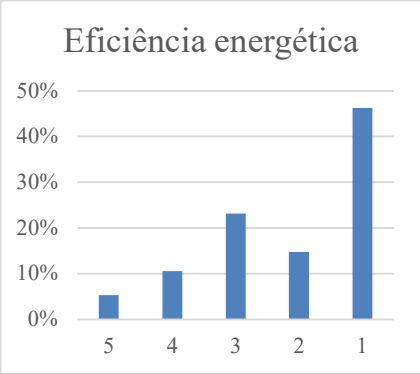


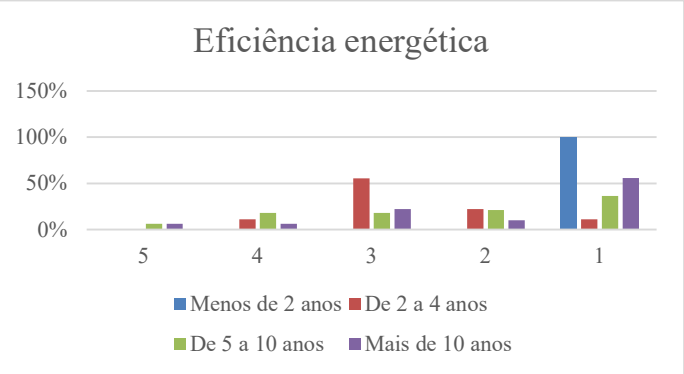


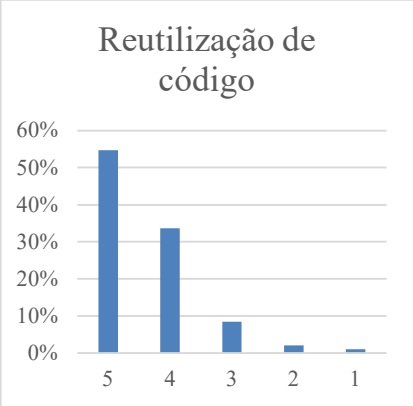
A.2. Resultados da seção 4

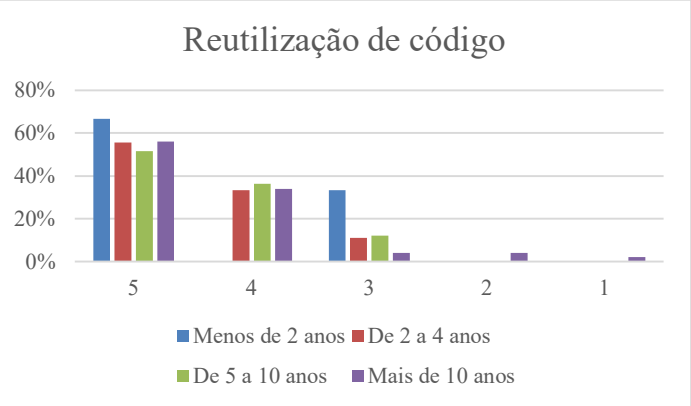
Ao decidir utilizar um determinado padrão de desenvolvimento, o quanto você leva em conta os impactos/ganhos abaixo com o que foi desenvolvido:

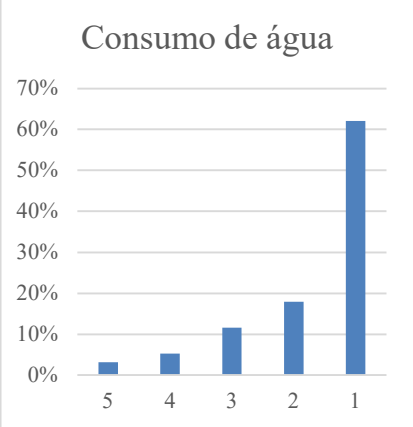
(Em uma escala de 1 a 5, sendo 1 não pensar sobre e 5 sempre pensar sobre esse impacto)

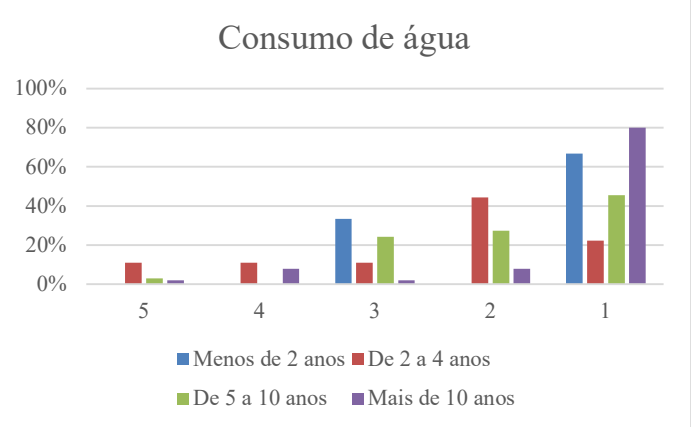

Eficiência energética
50%
40%
30%
20%
10%
0%
5
4
3
2
1

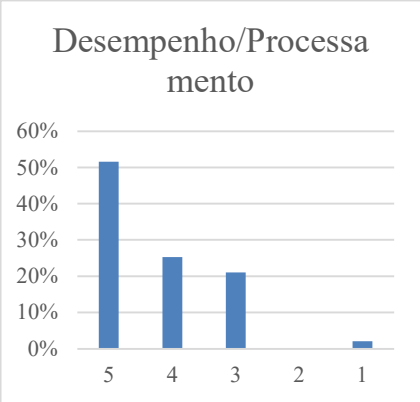

Eficiência energética
150%
100%
50%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

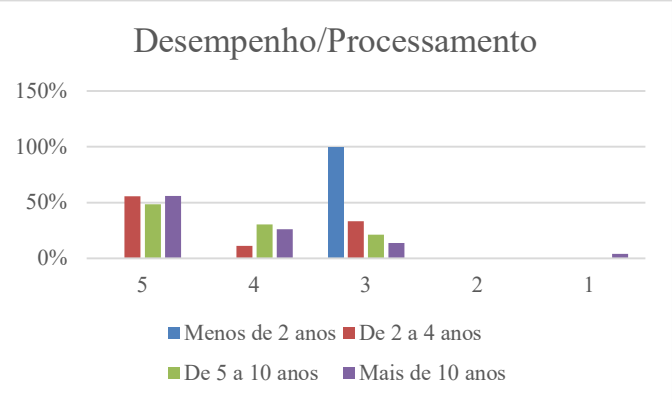

Reutilização de código
60%
50%
40%
30%
20%
10%
0%
5
4
3
2
1

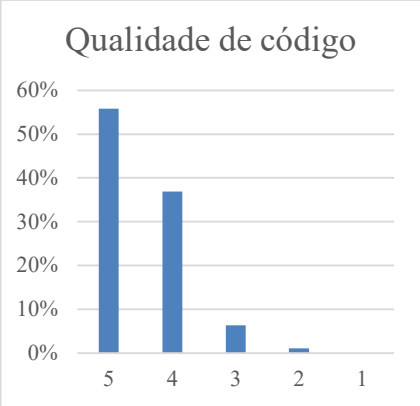

Reutilização de código
80%
60%
40%
20%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

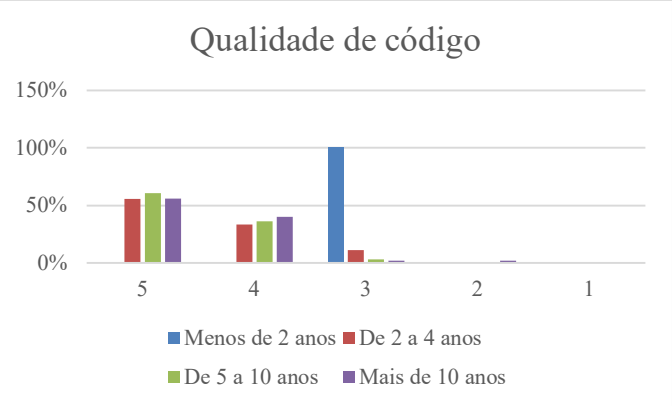

Consumo de água
70%
60%
50%
40%
30%
20%
10%
0%
5
4
3
2
1

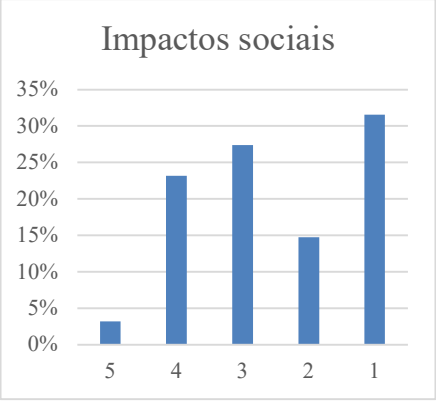

Consumo de água
100%
80%
60%
40%
20%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

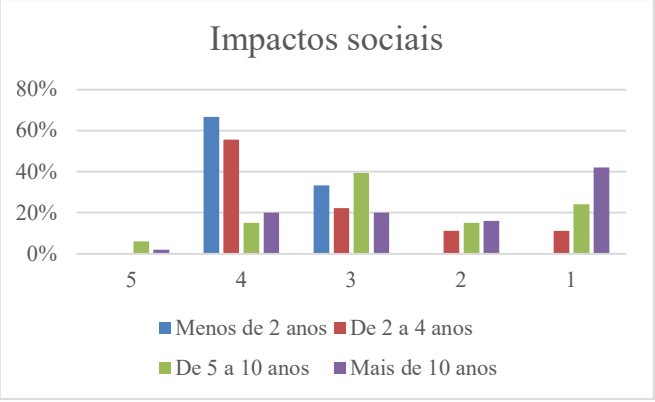

Desempenho/Processamento
60%
50%
40%
30%
20%
10%
0%
5
4
3
2
1

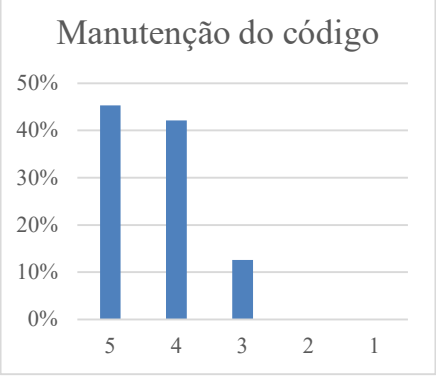

Desempenho/Processamento
150%
100%
50%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

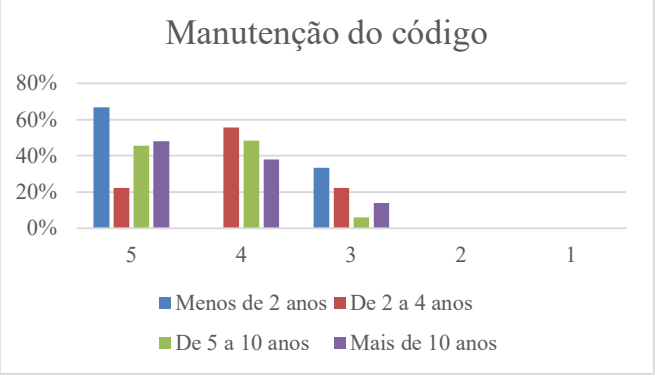

Qualidade de código
60%
50%
40%
30%
20%
10%
0%
5
4
3
2
1

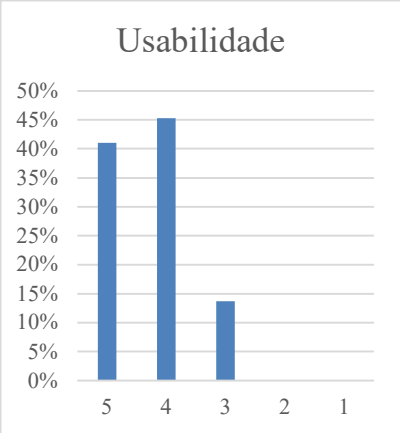

Qualidade de código
150%
100%
50%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

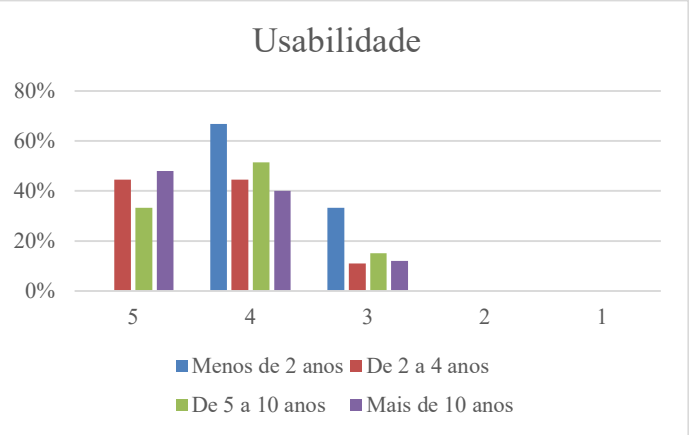

Impactos sociais
35%
30%
25%
20%
15%
10%
5%
0%
5
4
3
2
1

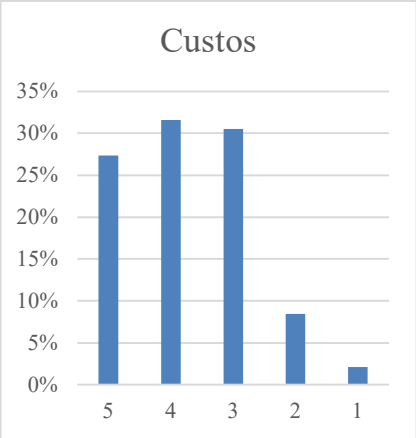

Impactos sociais
80%
60%
40%
20%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

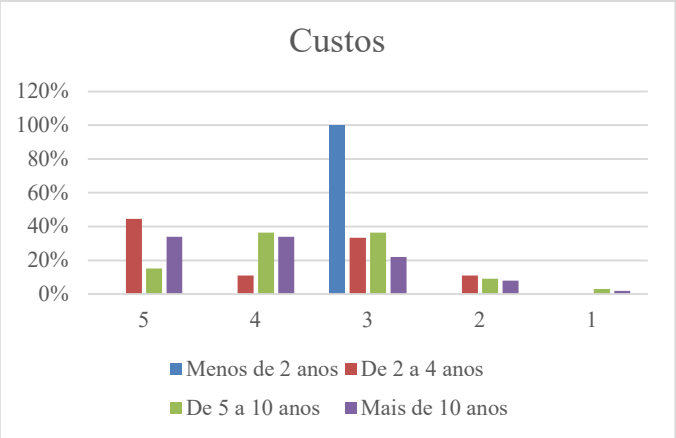

Manutenção do código
50%
40%
30%
20%
10%
0%
5
4
3
2
1

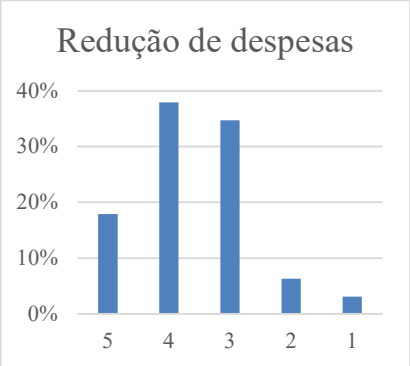

Manutenção do código
80%
60%
40%
20%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

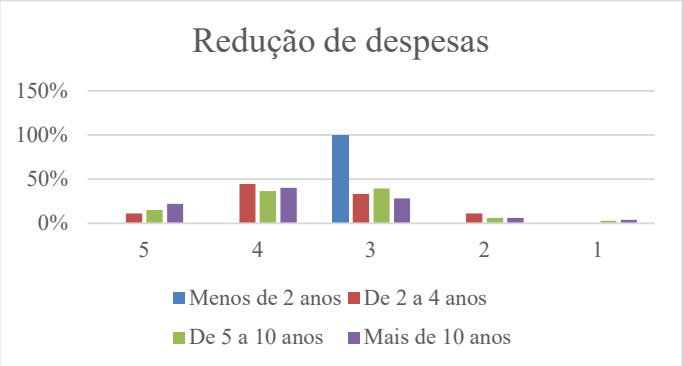

Usabilidade
50%
45%
40%
35%
30%
25%
20%
15%
10%
5%
0%
5
4
3
2
1

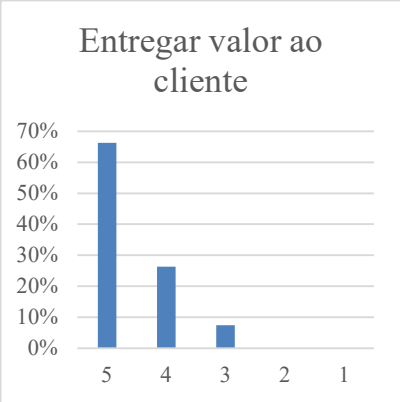

Usabilidade
80%
60%
40%
20%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

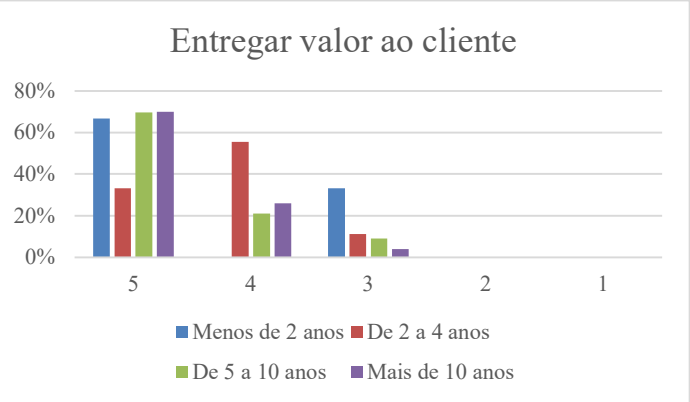

Custos
35%
30%
25%
20%
15%
10%
5%
0%
5
4
3
2
1

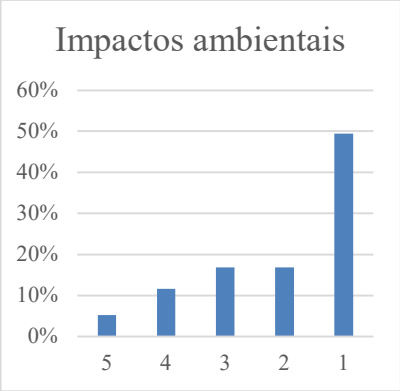

Custos
120%
100%
80%
60%
40%
20%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

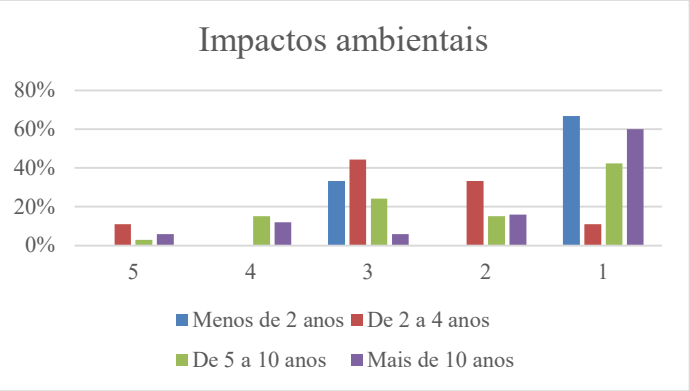

Redução de despesas
40%
30%
20%
10%
0%
5
4
3
2
1

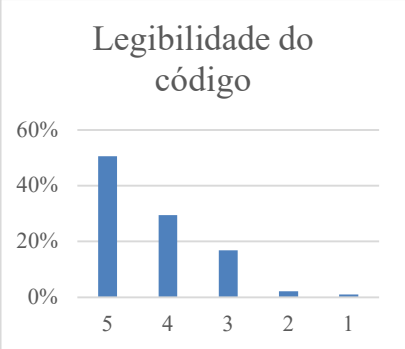

Redução de despesas
150%
100%
50%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

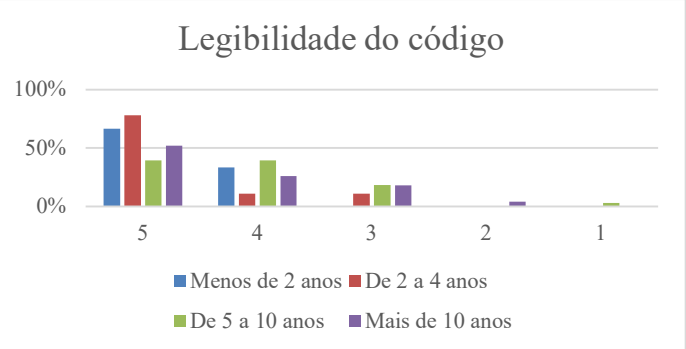

Entregar valor ao cliente
70%
60%
50%
40%
30%
20%
10%
0%
5
4
3
2
1


Entregar valor ao cliente
80%
60%
40%
20%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos


Impactos ambientais
60%
50%
40%
30%
20%
10%
0%
5
4
3
2
1


Impactos ambientais
80%
60%
40%
20%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos


Legibilidade do código
60%
40%
20%
0%
5
4
3
2
1


Legibilidade do código
100%
50%
0%
5
4
3
2
1
Menos de 2 anos
De 2 a 4 anos
De 5 a 10 anos
Mais de 10 anos

A.3. Formulário de Perguntas da Survey

Pesquisa - Desenvolvimento de Software Sustentável

Pesquisa realizada para identificar quais os princípios de programação conhecidos podem contribuir para se atingir o desenvolvimento de software sustentável.

*Obrigatório

1. Você trabalha ou já trabalhou com desenvolvimento de software? *

☐Sim

☐Não

2. A quanto tempo você trabalha com desenvolvimento de software? *

☐Menos de 2 anos

☐De 2 a 4 anos

☐De 5 a 10 anos

☐Mais de 10 anos

3. O quanto você avalia que conhece sobre os itens abaixo:

(Em uma escala de 1 a 5, sendo 1 nunca ter ouvido falar sobre e 5 conhecer e utilizar a técnica/padrão no seu dia a dia.)

Nunca ouvi falar	1	2	3	4	5 Conheço e utilizo
POO - Programação Orientada a Objetos *	O	O	O	O	O
SOA - Arquitetura orientada serviços *	O	O	O	O	O
Micro Serviços *	O	O	O	O	O
SOLID: Os 5 princípios da POO *	O	O	O	O	O
Clean Code *	O	O	O	O	O
GIT Flow *	O	O	O	O	O
Design Patterns *	O	O	O	O	O
DDD – Domain Driven Design *	O	O	O	O	O
Desenvolvimento de Software Ágil *	O	O	O	O	O
MVC - Multicamadas *	O	O	O	O	O

Rest / Restfull *	O	O	O	O	O
Software Sustentável *	O	O	O	O	O
Contêineres *	O	O	O	O	O

4. Ao decidir utilizar um determinado padrão de desenvolvimento, o quanto você leva em conta os impactos/ganhos abaixo com o que foi desenvolvido:
(Em uma escala de 1 a 5, sendo 1 não pensar sobre e 5 sempre pensar sobre esse impacto)

Nunca penso, quando desenvolvo	**1**	**2**	**3**	**4**	**5 Sempre penso, quando desenvolvo**
Eficiência energética *	O	O	O	O	O
Reutilização de código *	O	O	O	O	O
Consumo de água *	O	O	O	O	O
Desempenho/Processamento *	O	O	O	O	O
Qualidade de código *	O	O	O	O	O
Impactos sociais *	O	O	O	O	O
Manutenção do código *	O	O	O	O	O
Usabilidade *	O	O	O	O	O
Custos *	O	O	O	O	O
Redução de despesas *	O	O	O	O	O
Entregar valor ao cliente *	O	O	O	O	O
Impactos ambientais *	O	O	O	O	O
Legibilidade do código *	O	O	O	O	O