Análise da Utilização de Padrões de Projeto de Software em Sistemas Distribuídos, utilizando Arquitetura Limpa

Paulo F. P. Neto

[1]Bacharelado em Engenharia de Software
Instituto de Ciências Exatas e Informática – PUC Minas
Rua Cláudio Manoel, 1162 – Belo Horizonte – MG – Brasil

paulo.neto@sga.pucminas.br

Abstract. *The design patterns used are as a general solution to a recurring problem, possibility of code reuse and higher software quality. Patterns are often used in the implementation of software architecture, clean architecture has the SOLID Principles as one of its pillars. Currently there is a need for analysis on the knowledge and implementation of software developers related to the use of software architecture design patterns in distributed systems using clean. For this reason, we present an assessment of the knowledge and capability of software developers in relation to a model architecture focusing on clean architecture, systems design and distributed systems. It was obtained as an architecture that 75% of the cleaning principles were already more developed by all software developers 60% analyzed as a result of the work. by group found groups with different means of communication in relation to the ability to work with technologies.*

Keywords: design patterns, clean architeture, SOLID Principles

Resumo. *Os padrões de projetos são utilizados como uma solução geral para um problema recorrente, possibilitando reuso de código e uma maior qualidade do software. Padrões costumam ser utilizados na implementação de arquitetura de software, a arquitetura limpa tem os Princípios SOLID como um de seus princípios. Atualmente existe uma carência na análise sobre a capacidade de trabalho e de implementação dos desenvolvedores de softwares relacionado a utilização de padrões de projeto de software em Sistemas Distribuídos utilizando arquitetura limpa. Por essa razão, apresentamos uma avaliação do conhecimento e da capacidade de desenvolvedores de software em relação a implementação de arquitetura em camadas tendo foco na arquitetura limpa, padrões de projetos e sistemas distribuídos. Obteve-se como resultado que 75% dos princípios da arquitetura limpa, já foram desenvolvidos por mais de 60% dos desenvolvedores de softwares analisados neste trabalho.Concluiu-se que apesar da média de capacidade de trabalho dos desenvolvedores está acima de regular, quando agrupamos por grupo encontramos grupos com diferentes médias em relação a capacidade de trabalho com as tecnologias.*

Palavras-chave: padrões de projeto, arquitetura limpa, Princípios SOLID

1. Introdução

Padrões de projeto de software são utilizados com o propósito de oferecer uma solução para problemas recorrentes de um determinado sistema [Nazar and Aleti 2020]. Esses

padrões constituem um conjunto de artefatos que encapsulam o conhecimento dos problemas de projeto que ocorrem em um determinado contexto [Dwivedi et al. 2018]. Eles visam uma melhora na arquitetura e qualidade do software [Nazar and Aleti 2020]. Uma forma de se usar padrões, é na utilização de uma arquitetura limpa que é uma arquitetura que tem como um de seus princípios os princípios SOLID [Martin et al. 2018]. Os princípios SOLID consistem de um conjunto de cinco princípios que orientam a criação de estruturas que tolerem mudanças, e sejam base de componentes que podem ser usados em muitos sistemas de softwares [Martin et al. 2018], em particular, os sistemas distribuídos, que são sistemas cujos componentes estão localizados em computadores interligados em rede e que se comunicam e coordenam suas ações por troca de mensagens [Rehman et al. 2007].

Com a crescente demanda para criação de aplicações móveis a implementação de boas arquiteturas podem diminuir o esforço e o tempo necessários para a implementação e manutenção do software [BUENO 2021]. Devido a importância da arquitetura em projetos de software, este trabalho busca mapear o capacidade dos desenvolvedores de software de trabalharem com arquitetura em camadas, foco na arquitetura limpa, padrões SOLID e sistemas distribuídos.

A arquitetura limpa é usada no projeto do software a fim de facilitar a manutenção, testes e evolução do software através de um sistema de camadas com baixo grau de dependência [BUENO 2021]. Essa arquitetura possui oito princípios, que são eles: Regra de dependência, Princípio da abstração, Padrões SOLID, *Reuse/release equivalence principle* (REP), *Common closure principle* (CCP), *Common reuse principle* (CRP), *Acyclic dependency principle* (ADP), Portas e adaptadores [Martin et al. 2018]. Este trabalho trata de arquitetura limpa pois ela é derivada de outras arquiteturas existentes, como a Arquitetura em Cebola e a Arquitetura Hexagonal, que compartilham ideias semelhantes com a arquitetura limpa [BUENO 2021].

Dessa forma, o objetivo principal deste trabalho é avaliar a capacidade dos desenvolvedores de software de trabalharem com as tecnologias de arquitetura em camadas tendo foco na arquitetura limpa, padrões de projetos e sistemas distribuídos, tecnologias que se encontram em crescimento, visando também avaliar a capacidade do desenvolvedor em implementar os princípios da arquitetura limpa. Busca-se, assim, criar um mapeamento dessa capacidade de trabalho, relacionando com o tempo de experiencia dos desenvolvedores de softwares atuantes no mercado de trabalho sobre essas tecnologias. Para atingir esse objetivo, são propostos os seguintes objetivos específicos: i) Criar um questionário, relacionando as perguntas com os princípios da arquitetura limpa . ii) Analisar a correlação entre os níveis conhecimentos dos desenvolvedores nas diversas dimensões que envolvem a arquitetura limpa. iii) Identificar regras de associação para realizar análise dos dados coletados com o questionário.

Por meio da aplicação de um questionário, este trabalho buscou mapear como está a capacidade dos desenvolvedores de software de trabalharem com as tecnologias que este trabalho busca avaliar. Além de fazer uma correlação do capacidade dos desenvolvedores de software de trabalharem com as tecnologias em crescimento atualmente. É importante descobrir esse nível, para que se possa relacionar ele com o tempo de experiencia, e assim descobrir se o tempo de experiencia interfere na capacidade do desenvolvedor de trabalhar com determinada tecnologia. Deseja-se obter também uma relação entre as respostas,

através da aplicação do algoritmo de regra de associação para identificar relações causais entre os atributos arquiteturais de um software, no que diz respeito à sua utilização no mercado.

O restante deste trabalho está dividido em mais seis seções. A Seção 2 aborda a fundamentação teórica, aprofundando-se nos conceitos e teorias utilizadas neste estudo. A Seção 3 apresenta os artigos que possuem relação com o presente trabalho. A Seção 4 contempla a metodologia proposta no experimento. A Seção 5 apresenta os resultados obtidos. A Seção 6 discute os resultados e analisa as ameaças à validade do estudo. A Seção 7 apresenta os principais aspectos resultantes do trabalho e sugere potenciais trabalhos futuros.

2. Referencial Teórico

O referencial teórico apresentado nesta seção contém os três principais conceitos abordados pelo estudo: padrões de projeto de software, arquitetura limpa e sistemas distribuídos.

2.1. Arquitetura Limpa

A arquitetura limpa foi proposta como uma alternativa arquitetural que compartilha ideias de outros modelos de arquitetura em camadas como as Arquiteturas em Cebola e Hexagonal [Beltrão et al. 2020]. O objetivo dessa arquitetura é separar o software em camadas que se justapõem, com as regras de negócio da empresa na camada mais interna, seguidas pelos requisitos da camada de aplicação, enquanto as interfaces ficam nas camadas mais externas [Martin et al. 2018]. A arquitetura limpa foi apresentada contendo quatro camadas, como mostrado na Figura 1. Porém essa arquitetura não se limita às essas quatro camadas, mas permite a criação de novas camadas, desde que sigam a regra de dependência [Beltrão et al. 2020].

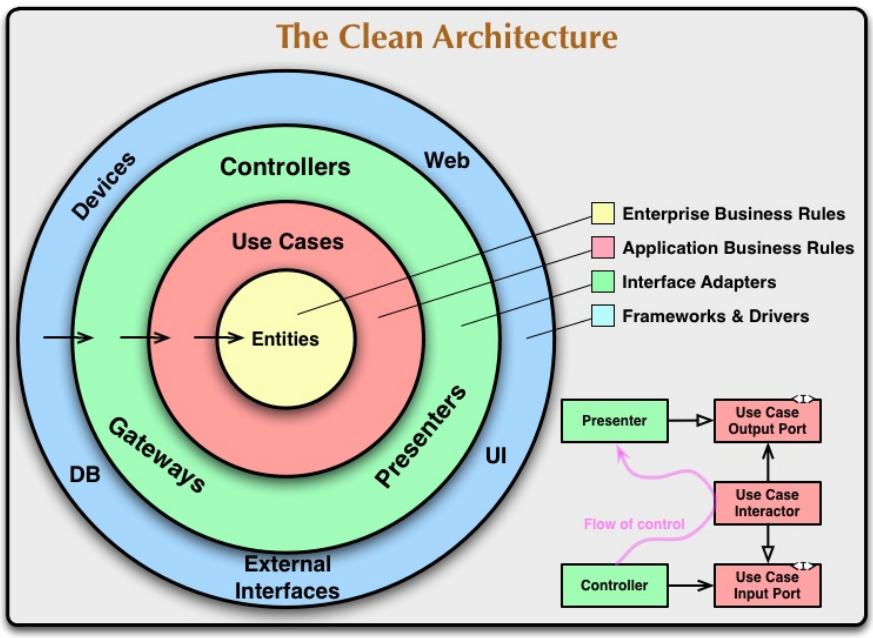

Figura 1. Arquitetura Limpa. Fonte: [Martin 2012]

Para implementar essa arquitetura é recomendado que se utilize princípios e padrões de desenvolvimentos, que são eles: a regra da dependência, princípio da abstração, Princípios SOLID, *Reuse/release equivalence principle* (REP), *Common closure principle* (CCP), *Common reuse principle* (CRP), *Acyclic dependency principle* (ADP), Portas e adaptadores.

2.1.1. Regra da dependência

Essa regra determina que "o nome de algo declarado em um círculo externo não deve ser mencionado pelo código em um círculo interno. Isso inclui funções, classes, variáveis ou qualquer outro nome ou entidade de software" [Martin et al. 2018]. As dependências de código só podem apontar para dentro.

Para permitir que os dados cruzem os limites de cada círculo, os desenvolvedores podem criar Objetos de Transferência de Dados (DTOs) [Beltrão et al. 2020], que são usados para transferir dados de um local a outro na aplicação, sem lógica de negócios em seus objetos. É comum utilizar para transferência de dados entre uma camada de visão e outra de persistência dos dados. Outra opção para passar dados pelas camadas é utilizar HashMap's, que são estruturas de dados que associam chaves de pesquisa a valores, tendo como objetivo a partir de uma chave simples, fazer uma busca rápida e obter o valor desejado [Martin et al. 2018].

2.1.2. Principio da abstração

O princípio da abstração é um dos princípios fundamentais da engenharia de software [Martin et al. 2018]. Ele afirma que a interface de um componente deve ser independente de sua implementação [Martin et al. 2018]. As regras de negócios e a camada de aplicação devem pertencer ao círculo mais interno, o mais abstrato. Enquanto as camadas adaptadoras e de infraestrutura devem conter detalhes de implementação, sendo localizadas no círculo mais externo o mais específico [Kanjilal 2020].

2.1.3. Common reuse principle

O *Common reuse principle* (CRP) ajuda a decidir quais classes e módulos devem ser colocados em um componente. De acordo com esse princípio, as classes e módulos que possuem a tendência de serem reutilizadas juntas devem estar em um mesmo componente, pois as classes raramente são reutilizadas isoladamente. O mais comum é que as classes reutilizáveis colaborem com outras classes que fazem parte da abstração reutilizável [Martin et al. 2018].

O CRP diz que se a classe pertence ao mesmo componente, este componente deve ter classes com várias dependências entre si. Por exemplo, uma classe *container* e seus iteradores associados. Essas classes são reutilizadas juntas porque estão fortemente acopladas entre si, e portanto devem estar no mesmo componente.

2.1.4. *Common closure principle*

O *Common closure principle* (CCP) diz que um componente não deve conter várias razões para mudar. Quando o código de uma aplicação precisa mudar, é preferível que todas as mudanças ocorram em um componente apenas em vez de serem distribuídas por vários componentes. Se as mudanças estão dentro de um único componente deve se implantar apenas o componente modificado, os demais componentes que não dependem dele, não precisam ser revalidados e nem reimplementados [Martin et al. 2018].

O CCP determina que todas as classes com probabilidades de mudarem pelos mesmo motivos estejam reunidas em um único componente. Esse princípio reduz o volume de trabalho relacionado ao processo de entrega. O CCP anota o termo fechado no mesmo sentido do OCP, que diz que as classes devem ser fechadas para modificações, mas abertas para extensões.

2.1.5. *Reuse/release equivalence principle*

Na última década, houve um grande aumento das ferramentas de gerenciamento de módulos, como *Maven*, *Leiningen* e *Ruby Version Manager* (RVM), uma vez que um grande número de componentes reusáveis e bibliotecas de componentes foram criados. O *Reuse/release equivalence principle* (REP) diz que componentes só devem ser reutilizados se puderem ser rastreados por meio de um processo de *release* e controle de versão [Martin et al. 2018].

Não é incomum que desenvolvedores de software sejam alertados de novos *releases* e com bases nas suas respectivas mudanças, decidam continuar utilizando o *release* anterior. Por isso é importante que o processo de *release* produza as devidas notificações e documentações para que os desenvolvedores de software tomem decisões informadas sobre o momento e a viabilidade para se utilizar a nova versão.

2.1.6. *Acyclic dependency principle*

Para podermos falar do *Acyclic dependency principle* (ADP), podemos considerar a seguinte situação, você trabalha o dia todo em algo, e faz com que ele funcione perfeitamente. No dia seguinte, entretanto, o funcionamento não está mais como deveria. Podemos chamar isso de “a síndrome da manhã seguinte”. Essa síndrome ocorre em ambientes onde muitos desenvolvedores modificam os mesmos arquivos-fonte. Sendo assim, não é incomum que uma equipe passe semanas sem que consiga criar uma *release* estável do projeto. O ADP apresenta uma possível solução para esse problema [Martin et al. 2018].

A ideia do ADP é particionar o ambiente de desenvolvimento em componentes passíveis de publicação. Esses componentes se tornam unidades de trabalhos de um desenvolvedor ou de uma equipe de desenvolvimento. Desta forma, quando um componente funciona, os desenvolvedores realizam a sua distribuição para que os outros possam utilizá-la. Na medida em que novas versões de um componente ficam disponíveis, outras equipes podem decidir se utilizam de imediato, ou continuarão com a versão anterior, caso não estejam preparados para receber as mudanças do novo *release*.

2.1.7. Portas e Adaptadores

A abordagem das portas e adaptadores tem como objetivo criar arquiteturas onde o código relativo às regras de negócio e ao domínio é independente e separado dos detalhes de implementação técnica como *frameworks* e bases de dados. As portas e adaptadores, tratam a web como apenas outro mecanismo de entrega. Nessa abordagem de portas e adaptadores se as interfaces possuem dependências de entradas de outros pacotes elas vão precisar ser declaradas como *public*. A abordagem mais simples utilizada para o código de portas e adaptadores é ter apenas duas árvores de código-fonte: Código de domínio (o "dentro"), Código de infraestrutura (o "fora") [Martin et al. 2018].

2.1.8. Princípios SOLID

Os princípios SOLID dizem como organizar as funções e estruturas de dados em classes e como essas classes devem ser interconectadas. O objetivo é a criação de estruturas de software que: tolere mudanças, seja fácil de entender e sejam a base de componentes que possam ser utilizados em sistemas de software [Martin et al. 2018]. Criado por Michael Fathers, SOLID representa um acrônimo de cinco princípios da programação orientada a objetos, teorizados por Robert C. Martins (Uncle Bob) por volta do ano 2000 [Magalhães and Tiosso 2019], os cinco princípios são:

2.1.8.1. *Single-responsibility Principle* O *Single-responsibility Principle* (SRP) diz que um módulo deve ter apenas uma razão para mudar. O SRP diz que um módulo deve ser responsável por um, e apenas um, ator. Quanto maior o número de responsabilidades de uma classe, maiores serão as chances de que essa classe venha a sofrer modificações no futuro e assim maiores serão as chances de inserções de *bugs* que podem danificar a classe por inteiro [Martin et al. 2018].

2.1.8.2. *Open-closed Principle* O *Open-closed Principle* (OCP) diz que um artefato de software deve ser aberto para extensão, mas fechado para modificação, ou seja, um artefato de software deve ter um comportamento que possa ser extensível mas sem sofrer modificações. Esse é o principal motivo de se estudar arquitetura de software, porque quando extensões simples nos requisitos forçam mudanças massivas no software, podemos dizer que os arquitetos desse sistema fracassaram. O OCP é visto como um princípio que guia o design de classes e módulos. A ideia é particionar o sistema em componentes e organizar esses componentes em uma hierarquia de dependência que proteja os componentes de nível mais alto das mudanças em componentes de nível mais baixo [Martin et al. 2018].

2.1.8.3. *Liskov Substitution Principle* Em 1988, Barbara Liskov escreveu que "O que se deseja aqui é algo como a seguinte propriedade de substituição: se para cada objeto o1 do tipo S existe um objeto o2 do tipo T, tal que, para todos os programas P definidos em termos de T, o comportamento de P fica inalterado quando o 1 é substituído por o2, então S é um subtipo de T." Em outras palavras, classes derivadas devem poder ser usadas no

lugar de classes base, sem prejuízo para a correção do programa [Magalhães and Tiosso 2019].

2.1.8.4. ***Interface Segregation Principle*** As interfaces são como classes e módulos, sendo assim elas também devem ser coesas, e terem uma única responsabilidade, possibilitando um maior reuso e evitando improvisações e soluções paliativas. [Aniche 2015].

Já Martin and Martin 2009, afirma que "o *Interface Segregation Principle* (ISP) reconhece que existem objetos que exigem interfaces não coesas; contudo, sugerem que os clientes não devem reconhecê-las como uma única classe. Em vez disso, os clientes devem reconhecer classes base abstratas que têm interfaces coesas."

2.1.8.5. ***Dependency Inversion Principle*** Conceitualmente o *Dependency Inversion Principle* (DIP) diz que módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações; abstrações não podem depender de detalhes; detalhes devem depender de abstrações [Magalhães and Tiosso 2019].

2.2. Sistema Distribuídos

Os sistemas distribuídos são um modelo da computação em que os componentes do sistema estão distribuídos em uma rede heterogênea [Bósio 2000] e se comunicam por troca de mensagens. Os sistemas distribuídos apresentam algumas vantagens devido à distribuição tais, como disponibilidade, desempenho e otimização de custos. Esses sistemas apresentam algumas características como concorrência de dados, falta de estado global, ocorrência de falhas parciais, heterogeneidade, assincronismo, evolução, autonomia e mobilidade [Brose et al. 2001]. Além de padrões de infraestrutura, preocuparam-se também em estabelecer especificações e recomendações que se fixaram como um Modelo Referencial para arquiteturas distribuídas, que tratam da forma como os componentes de um sistema distribuído são estruturados [Bósio 2000].

2.3. Regra de Associação

Uma regra de associação é uma expressão na forma de X $\rightarrow$ Y (lê-se: se X então Y), em que X e Y são ambos conjuntos de itens. Numa regra de associação é descrita pelo suporte do conjunto de itens que forma a regra, que é a porcentagem de vezes que aquele conjunto de itens aparece na base de conhecimento, e a confiança da regra, que é a porcentagem de vezes que Y acontece dado que X aconteceu. Dentre as medidas de interesse que permitem qualificar uma regra de associação, o *lift* indica o quanto a ocorrência da regra supera a probabilidade de coocorrência aleatória de seus itens individualmente. Essa medida permite identificar quais regras trazem novos conhecimentos, em comparação a regras que produzem conhecimento 'óbvio' [Barth 2006]. Para a identificação das regras de associação são definidos um suporte e uma confiança mínimos para que aquele conjunto seja considerado interessante.

3. Trabalhos Relacionados

Os trabalhos relacionados discutidos nesta seção envolvem as propostas de utilização de padrões de projetos em um sistema distribuído, a utilização desses padrões também é

estudada neste trabalho. Os demais artigos falam sobre a utilização dos padrões SOLID e a implementação de arquitetura limpa e seus princípios visando que precisamos entender sobre a utilização e implementação desses assuntos para que consigamos coletar os dados dos desenvolvedores de forma assertiva.

Burns and Oppenheimer (2016) têm como objetivo analisar os padrões de projeto para sistemas distribuídos baseados em contêineres. Eles descrevem três tipos de padrões de projeto em sistemas distribuídos baseados em contêiner: padrões de contêiner único para contêiner gerenciamento, padrões de nó único de contêineres em estreita cooperação e padrões de vários nós para distribuição de algoritmos. Os padrões para computação distribuída codificam melhores práticas, simplificam o desenvolvimento e tornam os sistemas onde eles são usados mais confiáveis. Esse estudo define que como nos sistemas orientados a objeto, os contêineres fornecem muitos dos mesmos benefícios, tornando mais fácil dividir a implementação entre várias equipes e reutilizar componentes em novos contextos [Burns and Oppenheimer 2016]. A relação com este trabalho acontece devido ao fato de que esse trabalho também realiza uma análise sobre padrões e sistemas distribuídos, porém é uma análise relativa ao conhecimento e experiência dos desenvolvedores de software.

Rachovski et al. (2022) haviam encontrado um problema relacionado a como escolher uma arquitetura mais adequada para minimizar as dependências do código e facilitar o desenvolvimento de novas funcionalidades. Sendo assim foi apresentada a arquitetura do aplicativo Mobiles Online, baseado no modelo arquitetura limpa. O objetivo principal é separar e eliminar as dependências entre os módulos de uma aplicação, utilizando dos princípios da arquitetura limpa para realizar essa separação entre o núcleo da aplicação de outros módulos. A relação com este trabalho é que além da utilização da arquitetura limpa também é utilizado e especificado os princípios da arquitetura limpa, e neste trabalho também analisamos a utilização e experiência sobre esses princípios por parte dos desenvolvedores de software.

Martin et al. (2018) tem como objetivo mostrar qual a maneira correta de se criar um código limpo, instruindo o leitor a ter noção da grande importância de se ter um código bem estruturado, organizado e de fácil leitura. Primeiramente mostrando o que é um código limpo e alguns princípios de boas práticas resultando no porquê o seu uso é de extrema importância, abordando a utilização de padrões de projeto SOLID para a construção de um código limpo. O trabalho conclui que desenvolvedores de softwares são autores de seu próprio código, sendo assim a responsabilidade de manter um código de boa qualidade é exclusivamente dele. A relação com este trabalho é a de como utilizamos os padrões SOLID, visto que ele é um dos princípios da arquitetura limpa.

Dantas (2021) tem como objetivo propor uma arquitetura de software limpa baseada na referência que permita o desenvolvimento modular dos sistemas de software, observadas as necessidades e padrões de projetos. A ideia do trabalho é aplicar a arquitetura limpa para o desenvolvimento de sistema de software em um sistema de microsserviços, separar o software em componentes, adicionando camadas arquiteturais bem definidas, gerenciando o acoplamento de recursos de hardware. A relação com este trabalho acontece, pois além da relação com a utilização de arquitetura limpa, eles utilizam um sistema de microsserviços que dialogam diretamente com a arquitetura dos sistemas distribuídos.

4. Materiais e Métodos

O estudo apresentado neste trabalho trata-se de uma pesquisa explotaria, utilizando o questionário como ferramenta de coleta de dados e o trabalho consiste em um único processo. Neste processo foi realizada a aplicação do questionário para a coleta de dados dos participantes. Este processo ocorreu por meio do questionário[1], que foi utilizado a fim de obter informações de profissionais de TI sobre seu conhecimento e experiências relacionadas a utilização de padrões de projeto de software, sistemas distribuídos e arquitetura limpa. Este instrumento de coleta de dados foi utilizado, pois ele alcança um grande número de pessoas geograficamente dispersas, isto o torna ideal para coletar uma grande quantidade de dados quantificáveis [Benyon 2011].

4.1. Procedimentos

O procedimento do processo foi dividido em quatro etapas, que são elas: elaboração do questionário, aplicação do questionário e coleta dos dados, agrupamento dos dados e análise dos dados.

Na primeira etapa foi realizada a elaboração do questionário. O questionário foi dividido em duas seções e desenvolvido no *Google Forms*. A primeira seção do questionário mostrou o Termo de Consentimento Livre e Espontâneo (TCLE), e verificou se os profissionais estavam aptos a participarem da pesquisa. Foi definido que apenas desenvolvedores de software estavam aptos a participarem. A segunda seção foi dividida em três pontos: no primeiro ponto foram coletadas informações sociodemográficas. Esses dados foram importantes para gerar artefatos por tempo de experiência e cargo. No segundo ponto foi utilizada Escala de *Linkert* para coletar informações sobre a capacidade dos participantes acerca dos assuntos abordados neste trabalho. No terceiro ponto os dados coletados eram informações sobre arquitetura limpa e foram feitas perguntas relacionadas aos oito princípios da arquitetura limpa, que são: Regra de Dependência, Princípio de abstração, SOLID, REP, PCC, CRP, ADP e portas e adaptadores. A Tabela 1 apresenta o relacionamento entre os pontos citados e as perguntas do questionário.

Tabela 1. Relacionando tópicos e número das perguntas

Tópicos	Número da Pergunta
Sociodemográficas	1 e 2
Conhecimento	3,4,5,6 e 9
Principios SOLID	7 e 8
Arquitetura Limpa	10 e 11
Regra de dependência	12 e 13
Principio da abstração	14
ADP	15
CRP	16
Portas e adaptadores	17
REP	18
PCC	19

A segunda etapa do processo consistiu na aplicação do questionário. O questionário foi publicado em grupos de mídias sociais focados na área de TI, grupo do Curso

[1]Questionário sobre o conhecimento e experiências dos desenvolvedores, disponível no Apêndice A

de engenharia de software, da equipe de TI da empresa Ambipar *Greentech*. As respostas do questionário foram coletadas por duas semanas e exportadas para um formato tabular no *Excel*. O questionário ficou disponivel por 3 semanas, e obteve-se 33 respostas de pessoas que estavam atuando em algum time de desenvolvimento de software, no momento em que o questionário foi respondido. Obteve-se um percentual de 100% de respostas válidas. O alto percentual de respostas válidas provavelmente está ligado à forma com que foi divulgado o questionário, focando em canais onde a população predominante é de desenvolvedores de software e assim foi possível acertar exatamente o público alvo desejado.

Na terceira etapa do processo os dados foram agrupados utilizando a ferramenta *Excel* para a geração de artefatos, contendo a média dos desenvolvedores primeiro em relação ao seu conhecimento sobre os cinco tópicos: Arquitetura de software, padrões de projeto, sistemas distribuídos, padrões SOLID e arquitetura limpa. Foi realizado esse cálculo geral e também foram agrupadas as respostas em quatro grupos com relação ao tempo de experiência. Foram utilizados os seguintes grupos: Até 2 anos, de 2 a 4 anos,de 4 a 6 anos e acima de 8 anos. Não foi utilizado o período de 6 a 8 anos pois só tínhamos um desenvolvedor que se encaixava nessa categoria. A Tabela 2 mostra o tempo de experiência relacionado com a quantidade de desenvolvedores.

Tabela 2. Quantidade de desenvolvedores por tempo de experiência

Tempo de experiência	Quantidade de desenvolvedores
Até 2 anos	6
de 2 a 4 anos	10
de 4 a 6 anos	8
de 6 a 8 anos	1
acima de 8 anos	8

Na quarta etapa foi calculada a porcentagem dos desenvolvedores em relação aos oito princípios da arquitetura limpa, que são eles: Regra de dependência, Princípio de abstração, Padrões SOLID, REP, PCC, CRP, ADP e Portas e adaptadores. Utilizou-se a relação entre tópicos e perguntas descritos na Tabela 1, para definir se uma pessoa utilizou determinado princípio. Para calcular a porcentagem de desenvolvedores que já haviam utilizado os princípios SOLID, aqueles que já implementaram pelo menos 3 dos 5 princípios SOLID e para calcular a porcentagem dos desenvolvedores que utilizam regra de dependência que haviam 2 perguntas relacionadas a esse princípio, foi considerado ele como tendo sido utilizado se o desenvolvedor respondeu sim na pergunta de número 12 e que já tenha utilizado algum objeto para transferência de dados entre as camadas.

Na quinta etapa foram utilizados os dados já agrupados pelo *Excel* para calcular o coeficiente de correlação de *Pearson* (r) entre as perguntas de conhecimento e experiências. As referências para esse cálculo são as seguintes, de 0.7 a 0.9 positivo ou negativo indica uma correlação forte. 0.5 a 0.7 positivo ou negativo indica uma correlação aceitável. 0.3 a 0.5 positivo ou negativo indica uma correlação fraca. 0 a 0.3 positivo ou negativo indica uma correlação insignificante.

Na sexta etapa foi utilizado o *RapidMiner*, que é uma plataforma de software de ciência de dados desenvolvida para fornecer um ambiente integrado para preparação de

dados, aprendizado de máquina, mineração de texto e análise preditiva, e foi utilizado para auxiliar no cálculo da regra de associação. A regra de associação tem como ideia encontrar elementos que resultam na presença de outros elementos em uma mesma transação, ou seja, encontrar padrões frequentes entre conjuntos de dados. Na Figura 2 mostramos como acontece o procedimento no *RapidMiner* até obtermos a regra de associação.

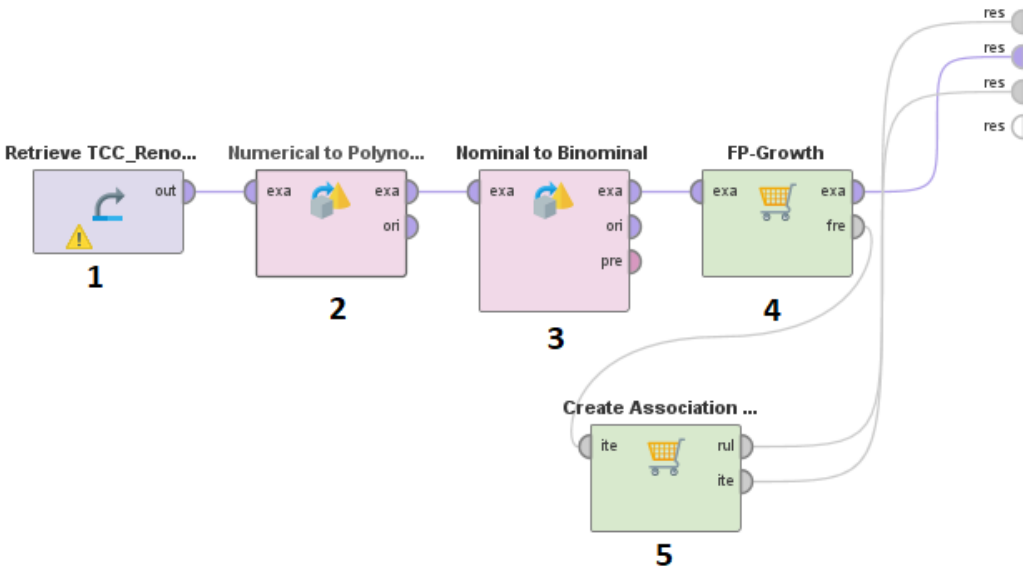

Figura 2. Processo Realizado pelo *RapidMiner* até a Geração da Regra de Associação

Foi realizada a numeração na Figura 2 dos passos para realizar o processo no *RapidMiner*, utilizou-se 5 etapas que foram: na primeira etapa foi utilizado o *excel* como uma base de dados e a partir dele foram realizadas as próximas etapas. Na segunda etapa foi transformado todos os valores numéricos obtidos como respostas para valores polinomiais. Na terceira etapa foi convertido todos os valores Nominais para Binomiais, essas conversões são realizadas para que se possa obter os seguintes valores mostrados na Figura 3 e para que assim seja realizada as próximas etapas. Na quarta etapa utilizou-se o *FP-Growth* para trazer o suporte de cada *Itemset*. Na quinta etapa é criada a regra de associação.

Row No. ↑	Nível em Sistema Distribuídos = 5	Nível em Sistema Distribuídos = 4
1	true	false
2	false	true
3	false	false
4	false	true
5	false	true
6	false	false

Figura 3. Transformação dos Dados do Questionário para cálculo da regra de associação

Pode-se observar na Figura 3 o que essa conversão realiza é transformação das

opções respostas numéricas para colunas e depois transforma suas respostas como *true* caso essa coluna tenha sido a resposta daquele desenvolvedor e caso não tenha sido ele coloca como *false*.

5. Resultados

Nesta seção, apresentam-se os resultados obtidos pelas análises dos dados no estudo proposto para compreender a Utilização de Padrões de Projeto de Software em Sistemas Distribuídos, utilizando Arquitetura Limpa. A seção trata sobre a análise dos resultados obtidos na aplicação da metodologia proposta.

5.1. Capacidade de trabalho dos desenvolvedores

A Tabela 3 mostra a capacidade de trabalho média dos desenvolvedores a respeito das tecnologias e abordagens relacionadas a este trabalho. Observa-se que nenhuma média de capacidade de trabalho foi inferior a 3. Contudo, a capacidade de trabalho em relação a arquitetura limpa apresenta a menor média. Por outro lado, a capacidade de trabalho em arquitetura de software apresenta a maior média.

Tabela 3. Distribuição do Valor Médio de Capacidade de trabalho dos Desenvolvedores em uma Escala de 1 (Muito Baixo) a 5 (Muito Alto).

Técnologia/Abordagem	Valor médio de conhecimento dos desenvolvedores
Padrões de Projeto	3.67
Arquitetura de software	3.73
Sistema Distribuídos	3.55
Princípios SOLID	3.45
Arquitetura Limpa	3.39

Na Figura 4, no eixo X, estão os 5 tipos de tecnologia analisados neste trabalho: Padrões de projeto; Arquitetura de Software; Sistemas Distribuídos; Princípios SOLID; Arquitetura Limpa. No eixo Y estão os valores médios da capacidade de trabalho dos participantes. E estão as 4 séries referentes a esse agrupamento relativo ao tempo de experiência: Até 2 anos; Entre 2 e 4 anos; Entre 4 e 6 anos; Acima de 8 anos. Observa-se que de acordo com o tempo de experiência a média da capacidade de trabalho aumenta com relação a cada tecnologia e abordagem estudada.

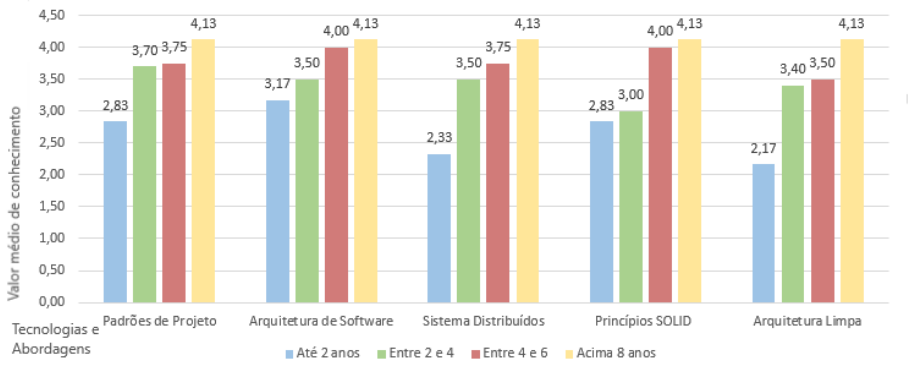

Figura 4. capacidade de Trabalho Média de Acordo com Tempo de Experiência

5.2. Implementação dos Princípios da Arquitetura Limpa

Na Figura 5, no eixo X, estão os 8 princípios da arquitetura limpa analisados neste trabalho: Princípios SOLID; Regra de dependência; Princípio da abstração; ADP; CRP; Portas e adaptadores; REP; PCC. No eixo Y estão as porcentagens de desenvolvedores que já implementaram aquele princípio. Nota-se que o princípio de portas e adaptadores foi implementado por mais desenvolvedores, por outro lado o CRP foi o princípio que menos desenvolvedores já implementaram.

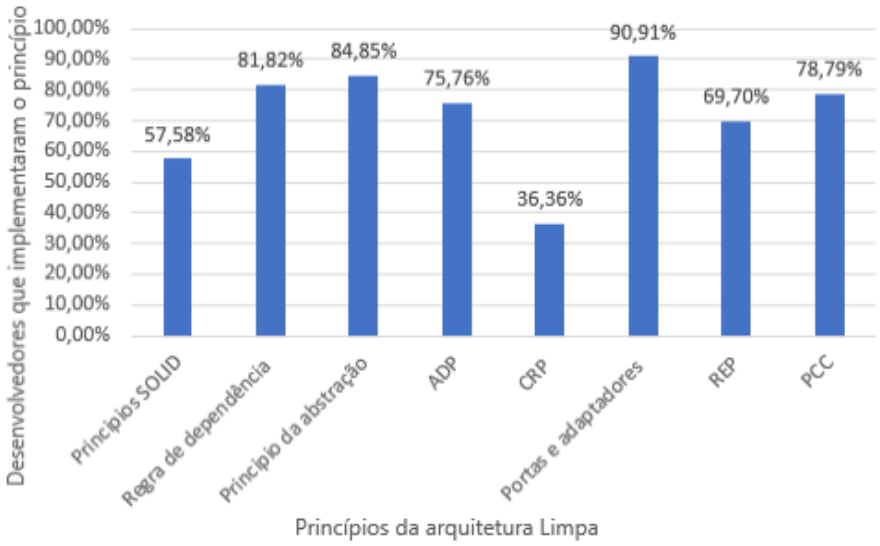

Figura 5. Percentual de desenvolvedores que implementaram determinado princípio

5.3. Correlação entre os fatores

Calculou-se o índice de correlação de *Pearson* (R) entre os 5 tipos de tecnologias analisadas neste trabalho e foram obtidas duas correlações positivas fortes, ou seja superiores a $R = 0.70$. Foram encontradas duas correlações consideradas fracas que se encontravam R entre 0.30 a 0.50, e uma considerada insignificante onde $R < 0,30$. Pode-se encontrar essas correlações na Tabela 4 abaixo:

Tabela 4. Correlação para as Capacidade de Trabalho dos Desenvolvedores em Relação as Tecnologias

	Sistema Distribuídos	*Padrões de Projeto*	*Arquitetura de Software*	*Padrões SOLID*	*Arquitetura Limpa*
Sistema Distribuídos	1.00				
Padrões de Projeto	0.58	1.00			
Arquitetura de Software	0.50	0.71	1.00		
Padrões SOLID	0.54	0.51	0.73	1.00	
Arquitetura Limpa	0.38	0.62	0.52	0.29	1.00

5.4. *FP-gowth*

Como resultado é possível ver na Figura 6, os *ItemSets* que possuem um suporte maior que o mínimo estabelecido na geração dos resultados. Foi definido como suporte mínimo 0.242, esse valor foi escolhido pois 0.242 em relação a quantidades de desenvolvedores que realizaram essa pesquisa, é uma frequência perto de 8 ou seja quase um quarto dos

Size ↓	Support	Item 1	Item 2	Item 3	Item 4
4	0.242	transferência de da...	Nível em Padrões d...	Cargo = Pleno	Nível em arquitetur...
3	0.303	transferência de da...	CRP	Nível em Padrões d...	
3	0.242	transferência de da...	CRP	Nível em Arquitetur...	
3	0.242	transferência de da...	CRP	Nível em arquitetur...	
3	0.242	transferência de da...	Nível em Padrões d...	Cargo = Pleno	
3	0.273	transferência de da...	Nível em Padrões d...	Nível em arquitetur...	
3	0.273	transferência de da...	Cargo = Pleno	Nível em arquitetur...	
3	0.242	transferência de da...	Nível em Padrões d...	Nível em arquitetur...	
3	0.242	CRP	Nível em Padrões d...	Nível em arquitetur...	
3	0.242	CRP	Nível em Padrões d...	Desenvolveu um si...	
3	0.242	CRP	Nível em Padrões d...	REP	
3	0.273	Nível em Padrões d...	Cargo = Pleno	Nível em arquitetur...	
2	0.576	transferência de da...	CRP		
2	0.394	transferência de da...	Nível em Padrões d...		
2	0.394	transferência de da...	Cargo = Pleno		

Figura 6. Tela do *RapidMiner* com os conjuntos de itens frequentes

dados obtidos. Outro motivo para a escolha desse suporte mínimo é que valores inferiores a esse darem quantidade conjuntos de *ItemSets* exponenciais.

Foram obtidos 69 *ItemSets* com suporte superior a 0.242. O maior conjunto obtido foi um único conjunto que possui quatro *Items*, com três *Items* obteve-se 11 conjuntos, com dois *Items* obteve 36 conjuntos e com um *Item* obteve 21 conjuntos. O maior suporte obtido foi de 0.879 e o maior em conjuntos de 2 ou mais *Items* foi de 0.576.

5.5. Regra de Associação

A Figura 7 apresenta as regras de associação, utilizando *lift* como critério de filtro. O *lift* de uma regra de associação de A para B indica que a probabilidade condicional de ocorrência da regra é superior à probabilidade da coocorrência aleatória de cada um dos itens individualmente [GONÇALVES 2007].

6. Discussão

De acordo com os dados encontrados e analisados neste artigo, é possível fazer algumas considerações sobre o objetivo geral desta pesquisa: “Avaliar o conhecimento e a capacidade dos desenvolvedores de software em relação a implementação de arquitetura em camadas tendo foco na arquitetura limpa, padrões de projetos e sistemas distribuídos”, definido no início deste artigo.

Com relação a capacidade de trabalho dos desenvolvedores de softwares atuantes no mercado de trabalho, analisados neste trabalho, quando se analisa a tabela 3, foi identificado que em média possuem um nível de conhecimento acima de regular. Em relação a assuntos mais genéricos como Arquitetura de Software, Sistemas Distribuídos e Padrões

Premises	Conclusion	Support	Confidence	Lift ↓
CRP, Desenvolveu um sistema distribuído	Nível em Padrões de Projeto = 3	0.242	1	2.538
Nível em Padrões de Projeto = 4, Cargo = Pleno	transferência de dados = DTOs, Nível em arquitetura de software = 4	0.242	0.889	2.444
Nível em Padrões de Projeto = 4, Cargo = Pleno	Nível em arquitetura de software = 4	0.273	1	2.357
transferência de dados = DTOs, Nível em Padrões de Projeto = 4, Cargo = Pleno	Nível em arquitetura de software = 4	0.242	1	2.357
CRP, Nível em arquitetura de software = 3	Nível em Padrões de Projeto = 3	0.242	0.889	2.256
CRP, REP	Nível em Padrões de Projeto = 3	0.242	0.889	2.256
Desenvolveu um sistema distribuído	CRP, Nível em Padrões de Projeto = 3	0.242	0.800	2.200
REP	CRP, Nível em Padrões de Projeto = 3	0.242	0.800	2.200
Desenvolveu arquitetura limpa	Nível em arquitetura de software = 3	0.242	0.727	2.182
Nível em arquitetura de software = 3	Desenvolveu arquitetura limpa	0.242	0.727	2.182
transferência de dados = DTOs, Nível em Padrões de Projeto = 3	Nível em arquitetura de software = 3	0.242	0.727	2.182
Nível em arquitetura de software = 3	transferência de dados = DTOs, Nível em Padrões de Projeto = 3	0.242	0.727	2.182
transferência de dados = DTOs, Nível em Padrões de Projeto = 4, Nível em arquitetura de software = 4	Cargo = Pleno	0.242	0.889	2.095
Nível em arquitetura de software = 3	Nível em Padrões de Projeto = 3	0.273	0.818	2.077
Desenvolveu um sistema distribuído	Nível em Padrões de Projeto = 3	0.242	0.800	2.031
REP	Nível em Padrões de Projeto = 3	0.242	0.800	2.031
transferência de dados = DTOs, Nível em arquitetura de software = 3	Nível em Padrões de Projeto = 3	0.242	0.800	2.031
Cargo = Pleno, Nível em arquitetura de software = 4	transferência de dados = DTOs, Nível em Padrões de Projeto = 4	0.242	0.800	2.031
Nível em arquitetura de software = 3	CRP, Nível em Padrões de Projeto = 3	0.242	0.727	2

Figura 7. Tela do *RapidMiner* com as regras de associação

de Projetos as médias se aproximam do nível alto de conhecimento. Quando se entra nas áreas mais específicas como princípios SOLID que estão dentro de padrões de projetos e arquitetura limpa que se encontra dentro da parte de arquitetura de software vemos esse nível de conhecimento se aproximar do regular.

Quando agrupa-se essa capacidade de trabalho de acordo com o tempo de experiência, mostrado na Figura 4 nota-se que a média capacidade de trabalho chega a 4.13, ou seja, eles possuem em média uma capacidade de trabalho alto em todas as tecnologias/abordagem. Os desenvolvedores de 4 a 6 anos possuem suas médias de capacidade de trabalho entre 3.5 e 4.0, ou seja, eles possuem os capacidade de trabalho mais próximos de alto do que de regular, mas ainda não se encontram no nível alto. Quando se trata dos desenvolvedores entre 2 e 4 anos de experiência nota-se que possuem a média dos suas capacidade de trabalho entre 3.0 a 3.7 mostrando assim que apesar de alguns média das capacidades estarem se aproximando do alto existem alguns pontos a melhorar. Quando analisa-se os desenvolvedores com menos de 2 anos de experiência temos as médias da capacidade de trabalho variando entre 2.17 e 3.17 ou seja, a média capacidade de trabalho varia entre baixo e regular.

Em relação aos princípios da arquitetura limpa, podemos notar que 75% dos princípios da arquitetura limpa, já foram desenvolvidos por mais de 60% dos desenvolvedores de softwares analisados neste trabalho. Os Princípios SOLID são implementados por 57.58% dos desenvolvedores e apenas o CRP com 36.36% é o único princípio, implementado por menos da metade dos desenvolvedores analisados neste artigo.

A regra de associação mostra que, apesar de mostrar baixas associações que atendem o suporte e confiança mínima estabelecidos e que envolvem como conclusão o desenvolvimento de arquitetura limpa, nota-se uma ocorrência de associações envolvendo os princípios da arquitetura limpa. Ou seja, apesar de não haver um grande número de desenvolvedores que desenvolveram arquitetura limpa, esses desenvolvedores utilizam dos

princípios da arquitetura limpa tendo assim um conhecimento que podem trazer facilidade e poucos pontos a serem aprendidos caso um dia precise implementar uma arquitetura limpa.

O trabalho desenvolvido possui limitações, assim como todo trabalho científico dessa natureza. A quantidade de pessoas para participar do experimento é limitada, devido ao trabalho ter como foco os desenvolvedores de software. Além disso, um número de participantes mais elevado poderia aumentar significativamente nas estatísticas dos resultados adquiridos. Além disso, devido ao tempo disponível e a disponibilidade de desenvolvedores não foi viável a realização de testes para que pudesse analisar a veracidade dos conhecimentos e experiências informadas.

7. Conclusão

Este estudo se propôs Avaliar o conhecimento e a capacidade dos desenvolvedores de software em relação a implementação de arquitetura em camadas tendo foco na arquitetura limpa, padrões de projetos e sistemas distribuídos. Observou-se os conhecimentos e experiências de desenvolvedores de software sobre Arquitetura de Software, Padrões de projeto, Sistemas Distribuídos, Arquitetura Limpa e Padrões SOLID, a partir de questionário.

Concluiu-se que apesar da média de capacidade de trabalho dos desenvolvedores está acima de regular, quando agrupamos por grupo encontramos o grupo Acima de 8 anos com uma média de capacidade de trabalho em 4.13. Entre 4 e 6 anos de experiência possuem uma média de capacidade de trabalho entre 3.5 e 4.0. Entre 2 e 4 anos de experiência possuem uma média de capacidade de trabalho entre 3.0 a 3.7. Abaixo de 2 anos de experiência possuem uma média de capacidade de trabalho entre 2.17 e 3.17. Conclui-se que apesar de muitos desenvolvedores não terem realizados implementação de uma arquitetura limpa de fato seguindo os princípios da arquitetura limpa, nota-se que eles já implementaram bastantes princípios obteve-se como resultado que 75% dos princípios da arquitetura limpa, já foram desenvolvidos por mais de 60% dos desenvolvedores de softwares analisados neste trabalho.

Para trabalhos futuros, indicamos utilização do índices de desempenhos nas empresas onde os desenvolvedores trabalham, para validação da veracidade das respostas informadas pelos desenvolvedores.

Referências

Aniche, M. (2015). *Orientação a Objetos e SOLID para Ninjas: Projetando classes flexíveis*. Editora Casa do Código.

Barth, F. J. (2006). Mineraçao de regras de associaçao em servidores web com rapidminer. *Disponível no site¡ http://fbarth. net. br/materiais/webMining/¿. Brasil*.

Beltrão, A. C., de Almeida Farzat, F., and Travassos, G. H. (2020). Technical debt: A clean architecture implementation. In *Anais Estendidos do XI Congresso Brasileiro de Software: Teoria e Prática*, pages 131–134. SBC.

Benyon, D. (2011). *Interação Humano-Computador*, volume 2. 2011.

Bósio, R. (2000). *Análise Comparativa Entre as Especificações de Objetos Distribuídos DCOM e CORBA Utilizando o Ambiente de Desenvolvimento DELPHI*. PhD thesis, Universidade Regional de Blumenau.

Brose, G., Vogel, A., and Duddy, K. (2001). *Java programming with CORBA: advanced techniques for building distributed applications*, volume 6. John Wiley & Sons.

BUENO, C. E. D. O. (2021). Desenvolvimento de um aplicativo utilizando o framework flutter e arquitetura limpa. *Trabalho de Conclusão de Curso*.

Burns, B. and Oppenheimer, D. (2016). Design patterns for container-based distributed systems. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.

Dwivedi, A. K., Tirkey, A., and Rath, S. K. (2018). Software design pattern mining using classification-based techniques. *Frontiers of Computer Science*, 12(5):908–922.

GONÇALVES, E. C. (2007). ata mining de regras de associação – parte 1. Disponível em: `http://www.devmedia.com.br/data-mining-de-regras-de-associacao-parte-1/6533`. (Acesso em: 13 maio 2022.

Kanjilal, J. (2020). A primer on the clean architecture pattern and its principles. Disponível em: `https://www.techtarget.com/searchapparchitecture/tip/A-primer-on-the-clean-architecture-pattern-and-its-principles`. (Acesso em: 13 maio 2022.

Magalhães, P. A. and Tiosso, F. (2019). Código limpo: padrões e técnicas no desenvolvimento de software. *Revista Interface Tecnológica*, 16(1):197–207.

Martin, R. and Martin, M. (2009). *Princípios, Padrões e Práticas Ágeis em C*. Bookman Editora.

Martin, R. C. (2012). The clean architecture. Technical report, The Clean Code Blog.

Martin, R. C., Grenning, J., Brown, S., Henney, K., and Gorman, J. (2018). *Clean architecture: a craftsman's guide to software structure and design*. Number s 31. Prentice Hall.

Nazar, N. and Aleti, A. (2020). Feature-based software design pattern detection. *arXiv preprint arXiv:2012.01708*.

Rehman, K., Stajano, F., and Coulouris, G. (2007). An architecture for interactive context-aware applications. *IEEE Pervasive Computing*, 6(1):73–80.

A. Questionário aplicado na coleta de dados

1. **Quantos anos você tem de experiência profissional na área de TI?**

   (a) Até 2 anos
   (b) de 2 a 4 anos
   (c) de 4 a 6 anos
   (d) de 6 a 8 anos
   (e) acima de 8 anos
   (f) Não possuo

2. **Qual seu cargo atual?**

   (a) Trainee
   (b) Júnior
   (c) Pleno
   (d) Sênior
   (e) Estagiário

3. **Qual seu conhecimento sobre Sistema Distribuídos?**

   (a) Muito Baixo
   (b) Baixo
   (c) Regular
   (d) Alto
   (e) Muito Alto

4. **Você já desenvolveu algum sistema distribuído?**

   (a) Sim
   (b) Não

5. **Qual seu conhecimento sobre Padrões de Projeto?**

   (a) Muito Baixo
   (b) Baixo
   (c) Regular
   (d) Alto
   (e) Muito Alto

6. **Qual o seu conhecimento sobre arquitetura de software?**

   (a) Muito Baixo
   (b) Baixo
   (c) Regular
   (d) Alto
   (e) Muito Alto

7. **Qual seu conhecimento sobre Padrões SOLID?**

   (a) Muito Baixo
   (b) Baixo
   (c) Regular
   (d) Alto
   (e) Muito Alto

8. **Quais princípios SOLID você já utilizou na implementação de softwares?**

   (a) Princípio da Responsabilidade Única
   (b) Princípio do Aberto/Fechado
   (c) Princípio de Substituição de Liskov
   (d) Princípio da Segregação de Interface
   (e) Princípio da Inversão de Dependência
   (f) Nenhum das alternativas acima

9. **Já utilizou arquitetura em camadas para o desenvolvimento de algum software?**

   (a) Sim
   (b) Não

10. **Qual seu conhecimento sobre Arquitetura Limpa?**

(a) Muito Baixo (c) Regular (e) Muito Alto
(b) Baixo (d) Alto

11. **Já utilizou arquitetura limpa para o desenvolvimento de algum software?**

(a) Sim (b) Não

12. **Como realizava a transferência de dados entre as camadas?**

(a) DTO (c) Outros
(b) HashMaps

13. **Você considera ter desenvolvido um software que atendia a regra de dependência?**

(a) Sim (b) Não

14. **O software desenvolvido separava a camada de regras de negócios da camada de interfaces?**

(a) Sim (b) Não

15. **Você já realizou o desenvolvimento de um sistema permitia que a regra de negocio seja testada sem a utilização de UI, banco de dados, servidor web ou qualquer outro elemento externo?**

(a) Sim (b) Não

16. **Você já desenvolveu algum software que tinha uma arquitetura que não dependia da existência de nenhuma biblioteca carregada de recursos?**

(a) Sim (b) Não

17. **Já realizou um desenvolvimento de um software onde a UI (Design de interface de usuário) poderia mudar facilmente, sem alterar o resto do sistema?**

(a) Sim (b) Não

18. **Algum software desenvolvido por você pode trocar Oracle ou SQL Server por um Mongo, BigTable ou CouchDB, entre outros, pois as suas regras de negócio não estão ligadas à base de dados?**

(a) Sim (b) Não

19. **Realizou o desenvolvimento de software onde as regras de negocio do software, não possuem conhecimento sobre as interfaces do mundo externo?**

(a) Sim (b) Não