VINICIUS TAVARES PIMENTA

TROCA DE BIBLIOTECAS EM SISTEMAS COM E SEM ARQUITETURA LIMPA: UMA ANÁLISE DE ESFORÇO

LAVRAS – MG

2023

VINICIUS TAVARES PIMENTA

TROCA DE BIBLIOTECAS EM SISTEMAS COM E SEM ARQUITETURA LIMPA: UMA ANÁLISE DE ESFORÇO

Artigo apresentado à Universidade Federal de Lavras, como parte das exigências do Programa de Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para a obtenção do título de Bacharel.

Prof. Dr. Ricardo Terra
Orientador

LAVRAS – MG
2023

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos
da Biblioteca Universitária da UFLA

Pimenta, Vinicius Tavares
Troca de Bibliotecas em Sistemas com e sem Arquitetura Limpa: Uma Análise de Esforço / Vinicius Tavares Pimenta. 1^a ed. – Lavras : UFLA, 2023.
36 p. : il.

Artigo (bacharel)–Universidade Federal de Lavras, 2023.
Orientador: Prof. Dr. Ricardo Terra.
Bibliografia.

1. TCC. 2. Monografia. 3. Dissertação. 4. Tese. 5. Trabalho Científico – Normas. I. Universidade Federal de Lavras. II. Título.

CDD-808.066

VINICIUS TAVARES PIMENTA

TROCA DE BIBLIOTECAS EM SISTEMAS COM E SEM ARQUITETURA LIMPA: UMA ANÁLISE DE ESFORÇO
REPLACING LIBRARIES IN ORDINARY SYSTEMS VERSUS THOSE FOLLOWING CLEAN ARCHITECTURE: AN EFFORT ANALYSIS

Artigo apresentado à Universidade Federal de Lavras, como parte das exigências do Programa de Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para a obtenção do título de Bacharel.

APROVADA em 24 de fevereiro de 2023.

Prof. Dr. Paulo Afonso Parreira Junior	UFLA
Prof. Dr. Antônio Maria Pereira de Resende	UFLA

Prof. Dr. Ricardo Terra
Orientador

LAVRAS – MG
2023

À minha família que sempre esteve ao meu lado.

AGRADECIMENTOS

Agradeço primeiramente a Deus a minha vida e saúde. Agradeço também aos meus pais, Sergio e Alda, por estarem presente ao meu lado durante a minha jornada acadêmica, apoiando nas minhas escolhas e fornecendo amparo nos momentos difíceis. Ao meu irmão, Thales, fonte de inspiração a continuar sempre buscando novos desafios.

À minha esposa, Heliomara, pelo incentivo, encorajamento e por todo o suporte emocional que me permitiram enfrentar e superar os desafios ao longo deste processo.

Agradeço ao meu orientador, Ricardo Terra, a orientação e os ensinamentos ao longo deste trabalho. Sua paciência e dedicação foram fundamentais para o meu crescimento e evolução durante o processo. O encorajamento que me proporcionou, incentivando-me a dar o meu melhor a cada dia, demonstrou a sua confiança na minha ideia e me motivou a seguir em frente.

E, por último, agradeço a formação oferecida pela Universidade Federal de Lavras.

RESUMO

A arquitetura limpa consiste em uma abordagem de desenvolvimento de software que busca facilitar a compreensão, manutenção e evolução do código. No entanto, a arquitetura limpa exige um uso demasiado de interfaces e classes adicionais para deixar o código mais desacoplado e coeso, o que aumenta a verbosidade e a complexidade do sistema. Diante disso, este trabalho de conclusão de curso avalia o esforço de trocas de bibliotecas em sistemas com e sem arquitetura limpa. Basicamente, um mesmo sistema desenvolvido em uma arquitetura convencional foi convertido usando todas as diretrizes e práticas da arquitetura limpa pelo primeiro autor deste artigo. As seguintes três bibliotecas foram substituídas: typeorm pela prisma, express pela apollo-server e bull pelo bullmq. O esforço das trocas foi medido em relação à alteração de linhas de código e ao tempo que o desenvolvedor levou para efetuar as trocas. Como as principais contribuições deste trabalho: (i) prover um dataset com dois sistemas de software equivalentes, um com arquitetura convencional e outro seguindo todos os princípios da arquitetura limpa; (ii) fornecer uma análise empírica dos esforços dos ajustes necessários para converter o sistema em arquitetura limpa; e (iii) conduzir um estudo empírico sobre o esforço para troca de bibliotecas em sistema com e sem arquitetura limpa. Como os principais resultados, o estudo conclui que as bibliotecas typeorm, apollo-server e bullmq agregam mais complexidade ao sistema em relação à quantidade de linhas de código em comparação com as bibliotecas prisma, express e bull, respectivamente. Ademais, verificou-se que o sistema com arquitetura limpa apresenta um menor esforço em termos de linhas de código para as trocas das bibliotecas prisma e apollo-server em comparação com o sistema convencional, com exceção a biblioteca bullmq em que foi indiferente para ambos sistemas. Por fim, constata-se que, embora o sistema com arquitetura limpa requisitou um tempo maior de implementação das bibliotecas, evidenciou-se um sistema com um acoplamento menor que o sistema convencional, devido ao fato de que as trocas de bibliotecas são mais rápidas, tornando-o mais fácil de manter.

Palavras-chave: Arquitetura de Software. Engenharia de Software.

ABSTRACT

Clean architecture is a software development approach that aims to facilitate code comprehension, maintenance, and evolution. However, clean architecture requires a significant use of interfaces and additional classes to make the code more decoupled and cohesive, which increases verbosity and system complexity. This B.Sc. conclusion paper evaluates the effort required to switch libraries in systems with and without clean architecture. Essentially, the same system developed using a conventional architecture was converted using all the guidelines and practices of clean architecture by the first author of this article. The following three libraries were replaced: typeorm with prisma, express with apollo-server, and bull with bullmq. The effort of the changes was measured in terms of changed lines of code and the time it took the developer to make the changes. The main contributions of this study are: (i) providing a dataset with two equivalent software systems, one with conventional architecture and the other following all the principles of clean architecture; (ii) providing an empirical analysis of the effort required to convert the system to clean architecture; and (iii) conducting an empirical study on the effort required to switch libraries in systems with and without clean architecture. The main results show that libraries typeorm, apollo-server, and bullmq add more complexity to the system in terms of lines of code compared to libraries prisma, express, and bull, respectively. Additionally, we found that the clean architecture system requires less effort in terms of lines of code to switch the prisma and apollo-server libraries compared to the conventional system, except for the bullmq library, where it was indifferent for both systems. Finally, we argue that although the clean architecture system required a longer implementation time for the libraries, it resulted in a system with lower coupling than the conventional system because the library switch itself was faster, making it easier to maintain.

Keywords: Software Architecture. Software Engineering.

Sumário

1 Introdução **7**

2 Arquitetura Limpa **8**

2.1 Camada de Entidades . . . 9

2.2 Camada de Casos de Uso . . . 10

2.3 Camada de Adaptadores de Interface . . . 10

2.4 Camada de *Frameworks* e *Drivers* . . . 11

2.5 Camada Principal e de Configuração . . . 12

3 Sistema **12**

3.1 Funcionalidades do Sistema . . . 12

3.2 Tecnologias e Funcionamento . . . 14

3.3 Metodologia . . . 16

3.4 Implementação Convencional . . . 16

3.5 Implementação com Arquitetura Limpa . . . 17

4 Análise do Esforço da Troca de Bibliotecas **20**

4.1 Metodologia . . . 21

4.2 Troca da biblioteca *typeorm* pelo *prisma* . . . 22

4.2.1 Implementação Arquitetura Limpa . . . 22

4.2.2 Implementação Convencional . . . 23

4.2.3 Análise Quantitativa . . . 24

4.3 Troca da biblioteca *express* pelo *apollo-server* . . . 25

4.3.1 Implementação Arquitetura Limpa . . . 26

4.3.2 Implementação Convencional . . . 27

4.3.3 Análise Quantitativa . . . 27

4.4 Troca da biblioteca *bull* pelo *bullmq* . . . 28

4.4.1 Implementação Arquitetura Limpa . . . 29

4.4.2 Implementação Convencional . . . 29

4.4.3 Análise Quantitativa . . . 30

4.5 Análise e Discussão . . . 31

5 Ameaças à validade **32**

6 Conclusão **32**

1. Introdução

A arquitetura de software corresponde à estrutura de um sistema, que se reflete na forma como os componentes estão dispostos, como eles se relacionam com seu ambiente e os princípios que norteiam seu *design* e evolução (IEEE, 2000). Nesse sentido, alguns modelos arquiteturais com propósitos distintos surgiram, como a arquitetura limpa, do inglês *clean architecture*, idealizado por Martin et al. (2018). Essa arquitetura em questão consiste em uma abordagem de desenvolvimento de software que visa facilitar a compreensão, manutenção e evolução do código, através da separação de responsabilidades em camadas claramente definidas e independentes, melhorando a testabilidade, a escalabilidade e a manutenibilidade (MARTIN et al., 2018).

No entanto, a arquitetura limpa aumenta a verbosidade e a complexidade do sistema devido ao demasiado uso de interfaces e classes adicionais para deixar o código mais desacoplado e coeso, além de demandar uma curva de aprendizado maior, especialmente para desenvolvedores acostumados com padrões menos complexos (MASOTTI, 2021). Diante desse cenário, este trabalho de conclusão de curso avalia o esforço de trocas de bibliotecas em sistemas com e sem arquitetura limpa. Basicamente, um mesmo sistema desenvolvido em uma arquitetura convencional (BRAZ, 2021) foi convertido usando todas as diretrizes e práticas da arquitetura limpa pelo primeiro autor deste artigo. Logo após, as seguintes três bibliotecas foram substituídas: *typeorm*[1] pela *prisma*[2], *express*[3] pela *apollo-server*[4] e *bull*[5] pelo *bullmq*[6]. O esforço das trocas foi medido em relação à alteração de linhas de código e ao tempo que o desenvolvedor leva para efetuar as trocas.

Como as principais contribuições deste trabalho: (i) prover um *dataset* com dois sistemas de software equivalentes, um com arquitetura convencional e outro seguindo todos os princípios da arquitetura limpa; (ii) fornecer uma análise empírica dos esforços dos ajustes necessários para converter o sistema em arquitetura limpa; e (iii) conduzir um estudo empírico sobre o esforço para a troca de bibliotecas em sistema com e sem arquitetura limpa. No que diz respeito aos principais resultados, o estudo conclui que as bibliotecas *typeorm*, *apollo-server* e *bullmq* agregam mais complexidade ao sistema em relação à quantidade de linhas de código em comparação com as bibliotecas *prisma*, *express* e *bull*, respectivamente. Também foi observado que a troca da biblioteca *prisma* gerou menos esforço no sistema com arquitetura limpa ao apresentar uma diminuição significativa na quantidade de linhas de código (-461 LOC vs. -331 LOC da implementação convencional). Da mesma forma, a troca da biblioteca *apollo-server* gerou menos esforço no sistema com arquitetura limpa, porém com um aumento expressivo na quantidade de linhas de código na implementação convencional (139 LOC vs. 51 LOC da implementação com arquitetura limpa). Por outro lado, a troca da biblioteca *bullmq* demostrou esforços equivalentes em ambos os sistemas, visto que a quantidade de linhas de código aumentaram igualmente. Por fim, verificou-se também que a implementação das bibliotecas no sistema com arquitetura limpa requereu mais esforço ao apresentar um tempo maior (*prisma*: 115 minutos vs. 105 minutos da implementação convencional, *apollo-server*:

[1] https://www.npmjs.com/package/typeorm
[2] https://www.npmjs.com/package/prisma
[3] https://www.npmjs.com/package/express
[4] https://www.apollographql.com/docs/apollo-server/
[5] https://www.npmjs.com/package/bull
[6] https://www.npmjs.com/package/bullmq

435 minutos vs. 420 minutos da implementação convencional e *bullmq*: 425 minutos vs. 420 minutos da implementação convencional). Contudo, a troca das bibliotecas no sistema com arquitetura limpa gerou um esforço menor ao demostrar um tempo inferior (*prisma*: 5 minutos vs. 40 minutos da implementação convencional, *apollo-server*: 5 minutos vs. 25 minutos da implementação convencional e *bullmq*: 5 minutos vs. 15 minutos da implementação convencional). Desse modo, conclui-se que, embora o sistema com arquitetura limpa requisitou um tempo maior de implementação das bibliotecas, evidenciou-se um sistema com um acoplamento menor que o sistema convencional, devido ao fato de que as trocas de bibliotecas são mais rápidas, tornando-o mais fácil de manter.

Este trabalho está organizado da seguinte forma. A Seção 2 introduz a arquitetura limpa. Em seguida, a Seção 3 define o sistema alvo utilizado, descreve a metodologia para o processo de conversão de um sistema convencional para arquitetura limpa e mostra a conversão do sistema convencional em arquitetura limpa. A Seção 4 descreve a metodologia para realização das trocas bibliotecas, efetua as substituições e apresenta as análises e as discussões de cada substituição. A Seção 5 descreve as ameaças à validade do estudo. Enfim, a Seção 6 relata os resultados obtidos e trabalhos futuros.

2. Arquitetura Limpa

A arquitetura limpa consiste em um padrão arquitetural baseado em um conjunto de arquiteturas desenvolvidas nas últimas décadas, como a Hexagonal (COCKBURN, 2005), Dados, Contexto e Interação (DCI) (REENSKAUG, 2009) e Fronteira, Controle e Entidade (BCE) (JACOBSON, 2004). Todas essas arquiteturas possuem similaridades e propósitos em comum. Portanto, a ideia de Martin et al. (2018), idealizadores da arquitetura limpa, foi de encapsular todas as características e conceitos dessas três arquiteturas de uma forma prática de modo a proporcionar ao software a independência de tecnologia, o reaproveitamento de código, alta coesão e melhor testabilidade.

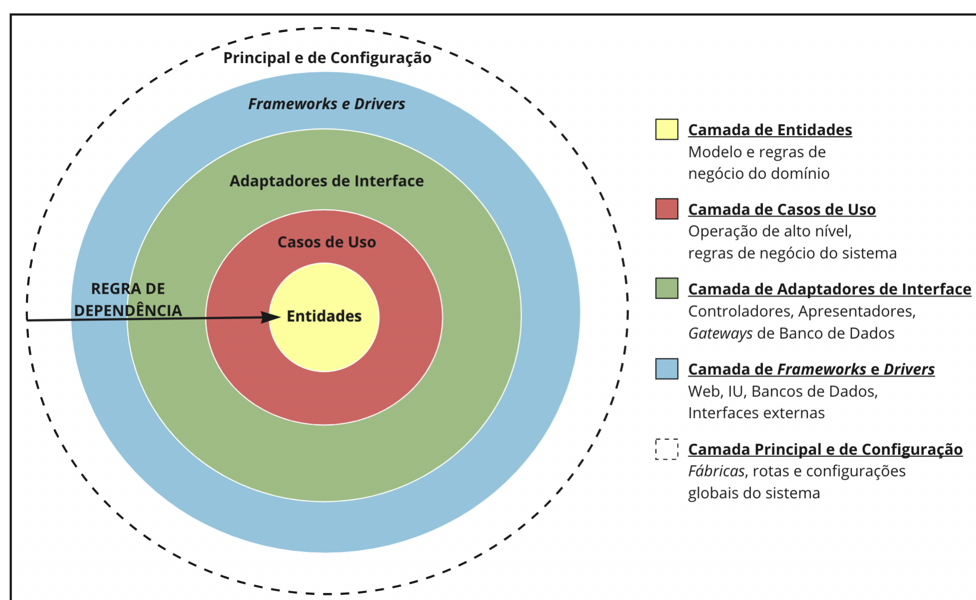

Figura 1. A Arquitetura Limpa (adaptado do trabalho de Martin et al. (2018))

A Figura 1 é uma representação da arquitetura limpa adaptada de Martin et al. (2018). Na imagem, é possível observar quatro camadas representadas por círcu-

los, além da seta, ilustrando a regra de dependência. Segundo o autor, as camadas internas não devem saber da existência das camadas externas, ou seja, as camadas externas devem depender das camadas internas e não o contrário. Dessa forma, manutenções nas camadas externas não devem refletir alterações nas camadas internas.

Na prática, são usados os princípios do SOLID para alcançar esses objetivo, o qual consiste em acrônimo para os seguintes cinco princípios da programação orientada a objetos que visa ajudar a criar sistemas flexíveis, robustos e escaláveis (OKTAFIANI; HENDRADJAYA, 2018):

- S (*Single Responsibility Principle*): Cada classe deve ter uma única responsabilidade e, portanto, uma única razão para mudar;
- O (*Open/Closed Principle*): As classes devem ser abertas para extensão, mas fechadas para modificação;
- L (*Liskov Substitution Principle*): Classes derivadas devem ser substituíveis por suas classes base;
- I (*Interface Segregation Principle*): Muitas interfaces específicas são melhores do que uma única interface genérica; e
- D (*Dependency Inversion Principle*): Dependa de abstrações, não de implementações concretas.

Em resumo, à medida que avança em direção às camadas mais internas, as políticas e o nível de abstração aumentam. Cada camada em específico desempenha um papel na arquitetura limpa. As próximas subseções são destinadas a explicar os detalhes de cada uma dessas camadas.

2.1. Camada de Entidades

Segundo Martin et al. (2018), a camada de Entidades é a região do software onde deve constar o nível mais alto de abstração do negócio, assim como deve concentrar as regras críticas de domínio. Além disso, conforme o autor, caso ocorra qualquer mudança em alguma camada externa, a camada de entidades não deve ser impactada ou propensa a manutenção.

Para exemplificar, considera-se o seguinte cenário: um usuário se cadastra em uma plataforma. Ao terminar o cadastro, a plataforma envia um e-mail de confirmação para o novo usuário com intuito de conceder o acesso ao sistema, caso o usuário seja o real proprietário do e-mail. Porém, se o sistema alterar essa regra de negócio para que conceda o acesso ao sistema mesmo o usuário não tendo confirmado o e-mail, a camada de entidades não deve estar sujeita a alteração. Isso se deve ao fato de que essa modificação está relacionada a regras de negócio da aplicação, localizadas na camada de Casos de Uso (mais detalhes na próxima subseção), ou seja, são regras que variam de aplicação para aplicação e, portanto, não configura uma regra de negócio de domínio. Dessa forma, uma modificação em uma camada externa não deve refletir mudanças na camada de entidades.

Entretanto, vale ressaltar que a adoção de um processo inverso ao exemplo anterior, isto é, um sistema anteriormente desprovido de solicitação prévia de confirmação via

e-mail passa a exigi-la, pode refletir em modificações nos arquivos pertencentes à camada de entidades, dependendo da solução adotada. Nesse cenário, a classe de usuário originalmente não incluía um atributo para verificar a validação do cadastro. Após a mudança no comportamento do sistema, o desenvolvedor pode optar por inserir um atributo na classe de usuário para lidar com a validação do cadastro.

Já uma classe *Email*, por exemplo, contendo todas as regras de negócio que validam a sintaxe do e-mail, ilustra um cenário localizado na camada de entidades, visto que o formato de um e-mail é restrito a regras críticas de domínio, ou seja, não é algo que varia entre as aplicações. Todo e-mail, independente da aplicação, deve seguir um conjunto de critérios para ser considerado um e-mail.

2.2. Camada de Casos de Uso

A camada de casos de uso, de acordo com Martin et al. (2018), é a região do software onde deve concentrar as regras de negócio da aplicação. É nessa camada onde os casos de uso que compõem todo o sistema são encapsulados e implementados. Esses casos de uso têm a responsabilidade de garantir com que as regras cruciais de negócio sejam atendidas por meio do direcionamento correto das entidades, assim como da orquestração exata dos fluxos de dados.

Do mesmo modo que a camada de entidades não pode ser propensa a manutenção caso tenha uma modificação em alguma camada mais externa, as demais camadas da arquitetura limpa seguem com essa mesma característica. A mudança de ORM[7], por exemplo, não deve refletir alterações na camada de casos de uso, visto que é um detalhe específico de implementação e não uma regra de negócio do sistema. Detalhes de implementação representam baixo nível de abstração e são localizadas em camadas mais externas da arquitetura limpa e, portanto, não devem causar manutenções nas camadas mais internas caso sofram alterações.

Já um cenário de caso de uso seria, por exemplo, um aluno só poder se matricular em uma disciplina apenas tendo cumprido os pré-requisitos da disciplina, visto que se trata de uma regra crucial de negócio da aplicação. Portanto, nessa camada deve ser realizado o direcionamento correto das regras críticas de negócio e a orquestração das entidades que representam o aluno, a disciplina e o diário de classe para que cumpra com o objetivo do caso de uso.

É importante ressaltar também que um caso de uso específico não está isento de manutenções quando algum outro caso de uso sofre alterações.

2.3. Camada de Adaptadores de Interface

A camada de Adaptadores de Interface é constituída por controladores, apresentadores e portas de entrada. Em resumo, conforme Martin et al. (2018), essa camada tem o papel de converter os dados em um formato mais apropriado para uso. Ainda sobre o objetivo dessa camada, Valente (2020) indica que “tem como função mediar a interação

[7]ORM (*Object-Relational Mapping*) é uma abordagem que consiste em tornar o desenvolvimento de software mais ágil por remover o distanciamento dos paradigmas entre a aplicação orientada a objetos e a base de dados relacional por meio do mapeamento de tuplas de uma relação do banco de dados em objetos e vice-versa (FONSECA, 2020).

entre a camada mais externa da arquitetura (sistemas complexos) e as camadas centrais (Casos de Uso e Entidades)".

Sobre as partes que pertencem à camada de adaptadores de interface, os controladores têm o papel de receber os dados da requisição e os adaptar para os casos de usos realizarem as operações. Já os apresentadores efetuam o processo contrário, ou seja, eles adaptam os dados retornados pelos casos de uso para os visualizadores. Por fim, as portas de entrada adaptam chamadas de algum agente externo, por exemplo, algum *framework* específico.

Um exemplo de adaptador de interface seria um adaptador de API *GraphQL*[8]. No caso, a responsabilidade desse adaptador é tanto receber os dados das requisições e direcioná-las para os controladores correspondentes quanto realizar o processo contrário, ou seja, receber o retorno dos controladores e transformá-los no formato apropriado para os visualizadores. Da mesma forma, caso o sistema altere para uma API REST[9], basta apenas implementar um adaptador, retirar o *GraphQL* e inserir o REST no sistema. Outra alternativa seria implementar um adaptador genérico para atender as requisições de ambas abordagens.

2.4. Camada de *Frameworks* e *Drivers*

Segundo Martin et al. (2018), a camada de *Frameworks* e *Drivers* é uma das camadas mais externas da arquitetura limpa, como pode ser observado na Figura 1. Ela normalmente contém *frameworks*[10], bibliotecas, serviços externos e dentre outras ferramentas. Essa amostra representa apenas detalhes de implementação e não deve afetar as camadas mais internas caso ocorra a alteração de uma das ferramentas.

Bibliotecas para se trabalhar com datas ou com identificadores únicos são uns dos casos da camada de *frameworks* e *drivers*. Para ilustrar, consideram-se as bibliotecas geradores UUIDs[11] como exemplo. Ao serem usadas na aplicação, elas devem ter métodos de gerar identificadores únicos e de verificar se uma *string* é um UUID válido. Portanto, caso ocorra a troca dessa biblioteca, o sistema deve continuar mantendo seu mesmo comportamento, visto que o papel das bibliotecas são detalhes de implementação e nada a mais, ou seja, as minúcias das amostras dessa camada não estão acopladas com as regras de negócios do sistema. Desse modo, o ato de inserir e remover coisas externas não deve ter efeito colateral nas camadas internas.

Outro exemplo é a troca de um Sistema Gerenciador de Banco de Dados (SGBD). Cada SGBD tem sua própria sintaxe de fazer as operações de busca e de escrita. Essas particularidades não têm interferência alguma nas regras de negócio do sistema. A forma de como são feitas as *queries* no *MySQL*[12] e no *PostgreSQL*[13] são apenas detalhes de implementação. Portanto, as camadas mais internas da arquitetura limpa não devem ser

[8]https://graphql.org/

[9]https://restfulapi.net/

[10]Um *framework* é uma estrutura pré-definida de componentes inter-relacionados que fornece uma abordagem padronizada para o desenvolvimento de aplicações, facilitando a resolução de problemas comuns e o desenvolvimento mais rápido e eficiente (BARRO, 2022).

[11]UUID (*Universally Unique Identifier*) é um identificador único com objetivo de garantir que não haverá conflitos com outros identificadores já existentes (FILHO, 2021).

[12]https://www.mysql.com/

[13]https://www.postgresql.org/

impactadas por manutenções caso ocorra a troca de SGBDs. Então, a injeção de dependência de outro banco de dados deve ser uma modificação simples de se fazer, além de não promover pontos de mudanças em arquivos de outras camadas quando o contexto se trata de um sistema que respeite as camadas arquiteturais e usa o princípio da inversão de dependência.

2.5. Camada Principal e de Configuração

A camada Principal e de Configuração, mencionada por Lemos (2021), se trata de uma região do software onde se reúne todos os módulos principais e de configuração do sistema, assim como é uma camada que detém de todo o conhecimento de cada minúcia de implementação da aplicação. Por exemplo, as diferentes formas de configurações para executar o sistema e dentre outras políticas de mais baixo nível da aplicação.

Outro caso são as fábricas[14] que também se encontram na camada principal e de configuração. Os códigos das fábricas de bibliotecas externas, de controladores e de repositórios[15] ficam desacopladas dos demais arquivos de outras camadas arquiteturais, tornando o software mais coeso.

3. Sistema

Neste artigo, adotou-se um sistema de barbearias desenvolvido por terceiros (BRAZ, 2021). Essa aplicação é projetada para auxiliar no gerenciamento de barbearias, permitindo a administração do fluxo de agendamento, solicitações e atendimentos dos serviços prestados (BRAZ, 2021). Em geral, a aplicação é utilizada para controlar a programação dos clientes com os prestadores de serviços que trabalham na barbearia (BRAZ, 2021). Ela permite que os clientes possam realizar agendamentos com seus prestadores de preferência diretamente pelo sistema, além de oferecer a possibilidade de visualizar a agenda de disponibilidade do prestador no momento da programação, contribuindo para a eficiência e conveniência do processo (BRAZ, 2021).

Para realização dos estudos, foi utilizada a implementação SEM DDD (chamada Implementação Convencional nesse artigo)[16]. Realizou-se uma modificação dessa implementação para um sistema que atende as características e conformidades arquiteturais da arquitetura limpa[17].

As Seções 3.1 e 3.2 são inspiradas no trabalho de Braz (2022).

3.1. Funcionalidades do Sistema

O sistema em discussão é voltado para o uso dos funcionários da barbearia (prestadores de serviço) e seus clientes, visando a programação de horários para a realização de determinado serviço. A Figura 2 ilustra o Diagrama de Caso de Uso da aplicação, destacando os dois atores envolvidos (Clientes e Prestadores de Serviço) e as funcionalidades disponíveis no sistema.

---

[14]As fábricas deslocam a responsabilidade de criar instâncias de objetos complexos e agregados para um objeto separado (EVANS; EVANS, 2004).

[15]https://martinfowler.com/eaaCatalog/repository.html

[16]https://github.com/lhleonardo/tcc-ufla

[17]https://github.com/vinicius-pimenta/tcc-ufla-vinicius-pimenta

A Figura 2 apresenta as funcionalidades disponíveis no sistema, descrevendo de forma geral as possibilidades de uso para a realização da gestão das tarefas. Essas funcionalidades são detalhadas a seguir.

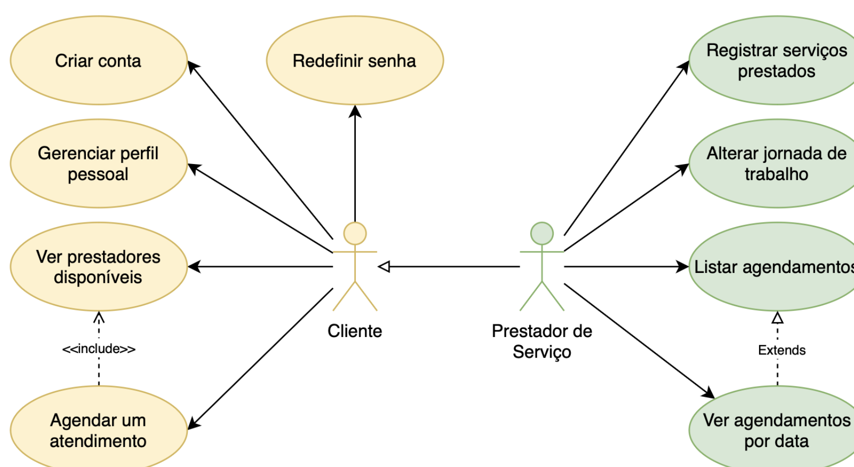

Figura 2. Diagrama de Casos de Uso. Fonte: do autor

Criar conta: A criação de conta é disponibilizada para ambos os atores do sistema, clientes e prestadores de serviço, e tem como objetivo realizar o cadastro de novos usuários no sistema. Essa funcionalidade é responsável por incluir os dados pessoais do usuário e possibilitar o acesso às demais funcionalidades da aplicação. No processo de cadastro, é necessário fornecer informações como nome completo, endereço de e-mail e senha, sendo o endereço de e-mail uma informação única na aplicação, impedindo valores duplicados. Ao finalizar a criação, o usuário é temporariamente bloqueado até que seja confirmada a conta, através do *link* de ativação enviado para o endereço de e-mail fornecido durante o cadastro. Para os prestadores de serviço, além das informações básicas necessárias para o cadastro de um cliente, também é requerida a informação da sua especialidade de atendimento.

Redefinir senha: A redefinição de senha é usada quando um usuário esquece a senha de sua conta. O processo de redefinir a senha inclui a verificação da identidade do usuário através de um endereço de e-mail. Depois de verificar a identidade, o usuário pode escolher uma nova senha e usá-la para acessar sua conta.

Gerenciar perfil pessoal: Após a conclusão do processo de criação de conta, os usuários têm acesso à funcionalidade de edição de suas informações cadastrais, permitindo a modificação desses dados. Além disso, é fornecida a opção de adicionar uma imagem de perfil, que é chamada de avatar no contexto do sistema, para personalizar a conta.

Ver prestadores disponíveis: Essa funcionalidade permite que os clientes acessem uma lista de todos os prestadores de serviço disponíveis para atendimento na barbearia, incluindo informações como imagem de perfil (avatar), nome completo e especialidade. Esse recurso permite ao usuário visualizar quais prestadores estão disponíveis para o atendimento na data selecionada, desde que o prestador possua pelo menos um horário disponível.

Agendar um atendimento: A funcionalidade de agendamento oferece aos clientes a possibilidade de programar um serviço com um determinado prestador selecionado. Para tal, o usuário deve selecionar o prestador de sua preferência e o horário desejado para realização do atendimento. É fundamental que o horário selecionado não esteja ocupado por outro agendamento e que esteja dentro da jornada de trabalho do prestador além de ser uma data válida (data futura).

Registrar serviços prestados: A funcionalidade de registro de serviços prestados é destinada para os prestadores de serviço da aplicação, com o objetivo de permitir que o prestador registre a finalização de um atendimento previamente agendado pelo cliente. Para tal, o prestador deve selecionar o agendamento que deseja registrar, seja por meio do cliente ou pela data de atendimento, e marcar como finalizado, permitindo assim o registro dos serviços executados.

Alterar jornada de trabalho: Essa funcionalidade fornece ao prestador de serviço a capacidade de editar suas informações de horários de atendimento na plataforma. Isso inclui a definição das horas dedicadas a agendamentos, duração de cada atendimento e o intervalo (se houver) entre cada atendimento. Essas informações são utilizadas para mostrar aos clientes os horários disponíveis para agendamentos, permitindo uma melhor programação e agendamento para o prestador e para os clientes.

Listar todos os agendamentos marcados: A funcionalidade de visualização de painel permite que o prestador de serviço tenha acesso a uma visão geral de todos os agendamentos programados para um dado mês, dentro de sua jornada de trabalho. Essa visualização mostra os dados dos clientes que realizaram as solicitações, incluindo o horário marcado para a data específica, possibilitando uma melhor gestão e planejamento para o prestador de serviços.

Ver agendamentos por data: A funcionalidade de visualização de agendamentos por data é semelhante à listagem dos agendamentos marcados, descritos anteriormente. Com essa funcionalidade, o prestador de serviço pode verificar os agendamentos registrados para um determinado dia, organizados por horário de agendamento e pelo cliente a ser atendido, o que permite uma gestão mais eficiente e planejamento diário para o prestador de serviços.

3.2. Tecnologias e Funcionamento

O sistema de barbearias foi desenvolvido de acordo com as técnicas de acesso a dados utilizadas em aplicações web, sendo classificado como uma API (*Application Programming Interface*) que fornece dados e valida as lógicas de negócio em um modelo centralizado e estruturado. O sistema adota o padrão REST (*REpresentational State Transfer*) para padronizar os estados da aplicação e segue o modelo RESTful[18].

A aplicação disponibiliza diferentes recursos acessíveis através de requisições HTTP, cada um relacionado a uma funcionalidade específica do sistema. A forma como essas requisições são nomeadas é descrita na Tabela 1. Cada recurso é descrito pelo seu conteúdo da requisição, o endereço de solicitação (URI) e o método HTTP apropriado

[18]O modelo RESTful é uma abordagem para o desenho de serviços que seguem a arquitetura REST (FIELDING, 2000), aderindo aos princípios de design desta arquitetura, e que possuem comportamentos específicos para cada tipo de requisição a um recurso.

para cada tipo de ação. Esses recursos estão relacionados com as funcionalidades específicas do sistema (ou funcionalidades auxiliares) descritas na Seção 3.1.

Tabela 1. Lista de *endpoints* da aplicação

Recurso	Método	Descrição
/users/	POST	Cadastra um novo usuário na API.
/users/avatar/	PATCH	Muda o *avatar* de um usuário.
/passwords/forgot/	POST	Envia e-mail de recuperação de senha.
/passwords/reset/	POST	Redefine a senha de acordo com *token* recebido por e-mail.
/sessions/	POST	Realiza autenticação na aplicação, retornando um *token* de acesso.
/profile/	GET	Busca informações do perfil do usuário autenticado.
/profile/	PUT	Atualiza informações de cadastro do usuário autenticado.
/appointments/	POST	Realiza um agendamento.
/appointments/:id	GET	Obtém informações de um determinado agendamento.
/appointments/:id/finish	POST	Marca um agendamento como efetivado.
/appointments/me/	GET	Mostra agendamentos do prestador autenticado em uma dada data.
/providers/:providerId/month-availability/	GET	Mostra os horários disponíveis de um prestador em um dado mês.
/providers/:providerId/day-availability/	GET	Mostra os horários disponíveis de um prestador em um dado dia.

Fonte: BRAZ (2021).

As implementações descritas foram desenvolvidas utilizando a linguagem *TypeScript*[19], um *superset* do *EcmaScript/JavaScript* amplamente utilizado para proporcionar tipagem parcial ao *JavaScript*. As tecnologias utilizadas na interface de programação de aplicativo (API) são descritas por meio das requisições recebidas por um servidor web, executado na plataforma *NodeJS*[20]. As requisições enviadas para a aplicação são interceptadas e processadas de acordo com a implementação do sistema.

Como parte da estrutura do sistema, as duas implementações em questão possuem mecanismos de armazenamento de dados que utilizam as seguintes tecnologias:

1. PostgreSQL: uma estrutura de persistência baseada em banco de dados relacional empregada para armazenar os registros gerenciados pelo sistema, incluindo tabelas, relacionamentos e atributos necessários;
2. MongoDB[21]: banco de dados NoSQL adotado para suprir as necessidades de armazenamento de dados temporários e notificações na aplicação, visando a disponibilidade de informações para a exibição aos usuários; e
3. Redis[22]: banco de dados no modelo chave/valor empregado como mecanismo de armazenamento de *cache* para consultas que demandam processamentos computacionais intensos.

Vale ressaltar que o primeiro autor deste artigo adicionou uma implementação de biblioteca de fila no caso de uso de "redefinir senha" para melhorar a experiência do usuário ao solicitar uma redefinição de senha, evitando possíveis congestões ou demoras no sistema. Além disso, o primeiro autor deste estudo inseriu tal biblioteca com propósito de ser usada no processo de substituição da Seção 4.4.

[19]https://www.typescriptlang.org/
[20]https://nodejs.org/en/
[21]https://www.mongodb.com
[22]https://redis.io

3.3. Metodologia

Para atingir os objetivos propostos nesta seção, foi realizado o seguinte conjunto de passos:

1. Foi utilizado um projeto de terceiros sujeito a mudanças para a arquitetura limpa. O sistema está localizado na base de dados *GitHub*[23];

2. Foi conduzido um estudo do código e criado um diagrama arquitetural correspondente. Em seguida, foram modelados os elementos presentes na arquitetura convencional de acordo com as camadas da arquitetura limpa, resultando em um novo diagrama arquitetural. A partir do novo diagrama e dos códigos, foram identificadas as decisões de projeto que não se alinham com os princípios estabelecidos pela arquitetura limpa;

3. Foram efetuadas as modificações e ajustes na implementação do projeto com a finalidade de corrigir as decisões de projeto e adequá-lo à arquitetura limpa, seguindo as diretrizes e boas práticas subjacentes a essa abordagem; e

4. Os esforços para os ajustes acima descritos foram analisados de forma empírica.

3.4. Implementação Convencional

A arquitetura do sistema em questão adota uma abordagem convencional, caracterizada pela clara separação de responsabilidades em diferentes componentes, tais como serviços, controladores, repositórios e bibliotecas externas, que são implementados em arquivos distintos. No entanto, é importante destacar que o sistema em questão apresenta algumas implementações voltadas à abstração, o que sugere um nível de complexidade e sofisticação acima do ordinário. Consequentemente, afirma-se que a arquitetura da implementação em questão é acima do que se considera razoável. Na Figura 3, é possível ter uma visão geral do projeto arquitetural do sistema em questão.

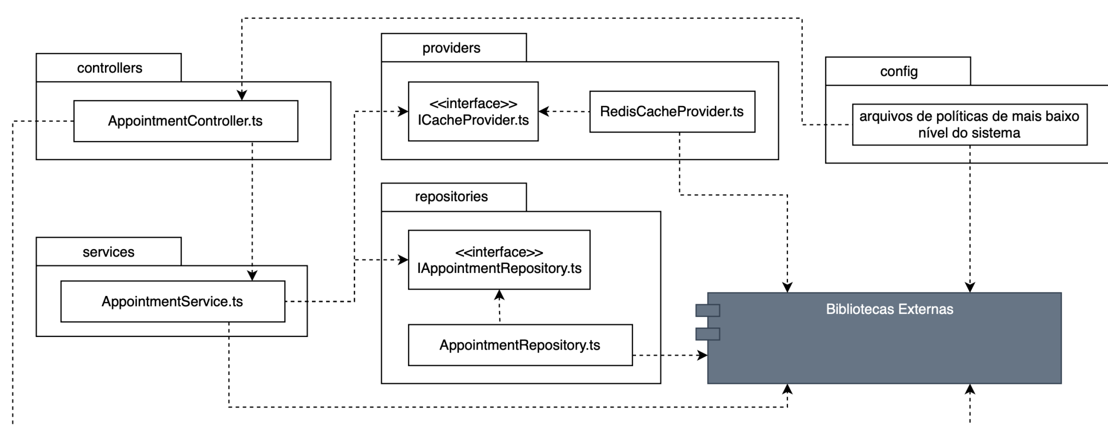

Figura 3. Diagrama Arquitetural da Arquitetura Convencional. Fonte: do autor

No entanto, é possível observar na Figura 3 que os arquivos `AppointmentService.ts` e `AppointmentController.ts` têm alto acoplamento

[23]https://github.com/

com as bibliotecas externas, o que pode resultar em um maior custo de manutenção. Isso pode ocorrer porque uma alteração ou troca em uma biblioteca externa poderia exigir ajustes em diversos arquivos. Por outro lado, a arquitetura convencional tende a ser menos verbosa por não necessitar de uma estrutura robusta que desacopla totalmente os códigos.

3.5. Implementação com Arquitetura Limpa

A primeira etapa do processo de modificação da arquitetura convencional para a arquitetura limpa consiste na tentativa de modelar os elementos presentes na arquitetura convencional nas camadas da arquitetura limpa, produzindo como resultado a Figura 4.

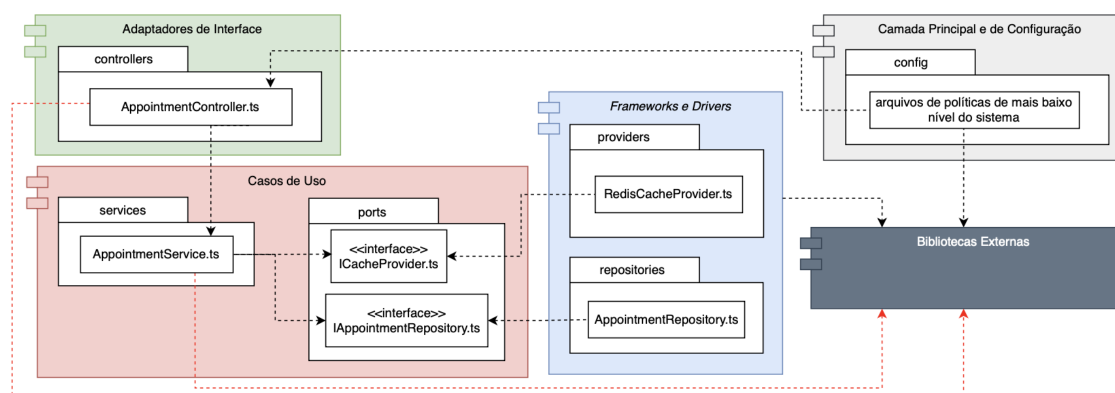

Figura 4. Diagrama Arquitetural da Arquitetura Convencional Representada na Arquitetura Limpa. Fonte: do autor

Em seguida, é realizada a identificação das decisões de projeto que não vão ao encontro do estabelecido pela arquitetura limpa através da Figura 4. As setas vermelhas da Figura 4 indicam decisões de projeto que divergem das propostas da arquitetura limpa. Importações de bibliotecas externas diretamente nas camadas de adaptadores de interface e casos de uso é uma dessas decisões de projeto. As primeiras três linhas do Código 1 evidencia as importações.

```
import { inject, injectable } from 'tsyringe';
import { format, getHours, isBefore, startOfHour } from 'date-fns';
import { classToClass } from 'class-transformer';

import AppError from '@errors/AppError';

import ICacheProvider from '@providers/cache/ICacheProvider';

import IAppointmentRepository from '@repositories/IAppointmentRepository';
import ICreateAppointmentDTO from '@repositories/dtos/ICreateAppointmentDTO';
import IFindAllAppointmentInDay from '@repositories/dtos/IFindAllAppointmentInDay';

import Appointment from '@models/Appointment';

@injectable()
class AppointmentService {
  constructor(
    @inject('AppointmentRepository')
    private appointmentRepository: IAppointmentRepository,

    @inject('CacheProvider')
    private cacheProvider: ICacheProvider,
```

```
23    ) {}
```

Código 1. AppointmentService.ts

A classe `AppointmentService` no Código 1 do projeto da arquitetura convencional faz uso direto da biblioteca *tsyringe*[24] na linha 1. Um dos objetivos dessa biblioteca no projeto é trazer praticidade no processo de injeção de dependências, resolução de dependências circulares e a garantia de apenas uma única instância da classe. Porém, o uso dessa biblioteca não vai ao encontro do estabelecido pela arquitetura limpa, visto que o *tsyringe* utiliza vários decoradores e contêineres espalhados nos códigos dos arquivos de várias camadas arquiteturais. Desse modo, para adequar as conformidades da arquitetura limpa, a modificação consiste remover essa biblioteca e codificar as próprias injeções de dependências através das fábricas. Além disso, é necessário implementar o padrão de projeto *Singleton*[25] para garantir a existência de uma única instância de uma classe. Esse processo de modificação gerou um alto custo, pois todos os arquivos das demais camadas da arquitetura limpa estavam acoplados com essa biblioteca externa, o que demandou um grande esforço para modificar o código.

Os usos diretos das bibliotecas externas *data-fns*[26] e *class-transform*[27] referenciados nas linhas 2 e 3 no Código 1 são outras decisões de projeto que não vão ao encontro do estabelecido pela arquitetura limpa. A alteração realizada da biblioteca *data-fns* consistiu em algumas etapas. A primeira delas criou uma camada de abstração, ou seja, um arquivo que assume um papel de contrato para que qualquer outra biblioteca com o mesmo propósito siga as definições impostas pelo contrato. A segunda etapa criou um arquivo que implementasse essa camada de abstração, contendo as codificações próprias dos métodos da biblioteca *date-fns*. Por fim, na última etapa, adotou-se a injeção de dependência no construtor do arquivo de casos de uso. Dessa forma, caso precise alterar a biblioteca externa, basta apenas criar um arquivo contendo as implementações dessa outra biblioteca externa, seguindo o contrato imposto pela camada de abstração. Após feito isso, é preciso apenas injetar essa nova dependência no construtor do arquivo desejado.

Já a alteração da biblioteca *class-transform* foi diferente. Como a biblioteca efetua algo simples, a estratégia utilizada removeu essa biblioteca e codificou um *mapper* próprio em um arquivo separado. Esse *mapper* é apenas uma classe com um método estático. Portanto, não foi necessário fazer a injeção de dependência, visto que o propósito do *mapper* é algo simples e não necessita de criar uma estrutura robusta. No arquivo de caso de uso foi realizada apenas a importação dessa classe e, consequentemente, seu uso no final do arquivo para retornar os dados da maneira desejada para os controladores.

É possível observar no Código 2 que não há agora importação direta de bibliotecas externas. Os detalhes dessas implementações foi delegada em arquivos específicos. Portanto, o código implementado dessa maneira tende a ser sustentável, visto que ele apresenta alta coesão e baixo acoplamento.

```
1  import AppError from '@shared/errors/app-error';
2
3  import ICacheProvider from '@external/cache/i-cache-provider';
```

[24] https://www.npmjs.com/package/tsyringe

[25] *Singleton* é um padrão de projeto com objetivo de garantir que uma classe tenha apenas uma instância, mantendo um ponto de acesso global para essa única instância (GAMMA et al., 1995).

[26] https://www.npmjs.com/package/date-fns

[27] https://www.npmjs.com/package/class-transformer

```
4  import IAppointmentRepository from '@external/repositories/i-appointment-repository'
       ;
5  import ICreateAppointmentDTO from '@external/repositories/dtos/i-create-appointment'
       ;
6  import IDateProvider from '@external/date/i-date-provider';
7
8  import Appointment from '@entities/appointments';
9
10 class CreateAppointmentUseCase {
11   private appointmentRepository: IAppointmentRepository;
12   private cacheProvider: ICacheProvider;
13   private dateProvider: IDateProvider;
14
15   constructor(
16     appointmentRepository: IAppointmentRepository,
17     cacheProvider: ICacheProvider,
18     dateProvider: IDateProvider
19   ) {
20     this.appointmentRepository = appointmentRepository;
21     this.cacheProvider = cacheProvider;
22     this.dateProvider = dateProvider;
23   }
```

Código 2. create-appointment-useCase.ts

Por fim, tem-se também a ausência do arquivo da camada de entidades como a última decisão de projeto que não vai ao encontro ao estabelecido pela arquitetura limpa. Dessa maneira, foi-se necessário fazer a criação desse arquivo, como também realizar uso dessa entidade nos arquivos de casos de uso para validar as regras de domínio. Como pode ser observado nas linhas 20 a 24 do Código 3, há regras que verificam a existência de valores para os atributos de data e de identificador único. Desse modo, somente após a aceitação da construção dessa classe que é chamado os arquivos do ORM para continuidade ao fluxo do caso de uso.

```
1  type AppointmentProps = {
2    providerId: string;
3    userId: string;
4    date: Date;
5    createdAt: Date;
6    updatedAt: Date;
7  };
8
9  export class AppointmentEntity extends Entity<AppointmentProps> {
10   private constructor(props: AppointmentProps, id?: string) {
11     super(props, id);
12   }
13
14   ...
15
16   public static create(
17     props: AppointmentProps,
18     id?: string,
19   ): AppointmentEntity {
20     if (props.providerId == null) throw new AppError('providerId field is mandatory'
         );
21     if (!props.userId) throw new AppError('userId field is mandatory');
22     if (!props.date) throw new AppError('date field is mandatory');
23     if (!props.createdAt) throw new AppError('createdAt field is mandatory');
24     if (!props.updatedAt) throw new AppError('updatedAt field is mandatory');
25
26     const appointment = new AppointmentEntity(
27       {
28         providerId: props.providerId,
29         userId: props.userId,
30         date: props.date,
31         createdAt: props.createdAt,
```

```
32            updatedAt: props.updatedAt,
33          },
34          id,
35        );
36
37        return appointment;
38      }
39    }
```

Código 3. appointment.ts

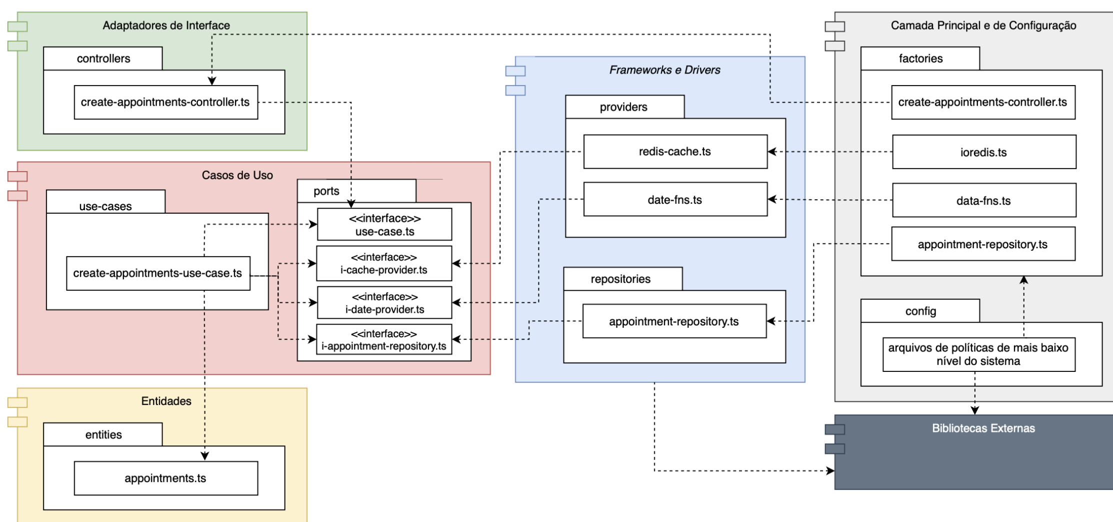

Figura 5. Diagrama Arquitetural do Código Alterado. Fonte: do autor

Para fins de análise, comparação e/ou replicabilidade do estudo, é importante mencionar que o código completo desse projeto – incluindo *providers* e repositórios – está disponível em repositório público[28].

Na Figura 5, é possível ter uma visão ampla de como a arquitetura do sistema ficou após o processo de modificação. Como pode ser observado, há uma divisão clara de cada uma das camadas da arquitetura limpa e suas dependências. É constatado que não há nenhuma decisão de projeto que não vai ao encontro estabelecido pela arquitetura limpa.

4. Análise do Esforço da Troca de Bibliotecas

As alterações em um sistema de software acontecem para sanar erros existentes ou para incluir novas funcionalidades e recursos (TRIPATHY; NAIK, 2014). Um exemplo recente de correção de software aconteceu em 2021, em que foi descoberta uma falha de segurança no Log4j, biblioteca *open source* que visa fazer processos de auditoria (OLIVEIRA, 2021). A vulnerabilidade do Log4j permite o registro de uma *string* específica que força o sistema a executar um *script* malicioso, o que pode conceder ao atacante o controle total, roubando dados, instalando *malwares* e realizando todo tipo de dano possível ao sistema (OLIVEIRA, 2021).

Portanto, esta seção visa fazer uma análise do esforço da troca de bibliotecas em um sistema convencional e em um com arquitetura limpa com propósito de fornecer dados

[28]https://github.com/vinicius-pimenta/tcc-ufla-vinicius-pimenta

que possam ser usados em tomadas de decisões por profissionais da área de desenvolvimento de software.

Esta seção está organizada da seguinte forma. A Seção 4.1 descreve a metodologia e as decisões tomadas. As Seções 4.2, 4.3 e 4.4 implementam e comparam as bibliotecas nos sistemas de software, sendo cada uma das seções destinada a uma única biblioteca. Enfim, a Seção 4.5 discute os resultados.

4.1. Metodologia

Para atingir o objetivo proposto nesta seção, foi realizado o seguinte conjunto de passos:

1. O autor deste TCC conduz em sessões de uma hora consecutiva por dia para a substituição de bibliotecas. Essa metodologia foi adotada para evitar possíveis erros de medição devido a fatores como fadiga e cansaço. A substituição foi inicialmente realizada no sistema com arquitetura limpa, seguida do sistema convencional, com o objetivo de minimizar qualquer possível viés em favor da arquitetura limpa. Após a conclusão das substituições, as funcionalidades foram verificadas para garantir o sucesso das operações.

2. Foi utilizada a SonarQube[29], uma ferramenta *open source* que realiza a inspeção de código com o objetivo de coletar dados para análise. Dentre as várias opções de dados fornecidas pela ferramenta, as métricas selecionadas para a análise foram:

   (a) Linhas de código: Conforme Fenton (1999), a métrica de linhas de código (LOC) fornece uma visão geral da complexidade e legibilidade do código;

   (b) Número de Funções: De acordo com Chidamber e Kemerer (1994), a medida do número de funções é fundamental para determinar a modularidade e divisão de tarefas do código;

   (c) Número de Classes: Segundo Chidamber e Kemerer (1994), a quantidade do número de classes é fundamental para avaliar a estruturação e divisão de tarefas do código; e

   (d) Número de Arquivos: Com base em Henry e Kafura (1981), a medida do número de arquivos é essencial para analisar a distribuição e organização do código.

3. Análise dos resultados: Embora existam essas métricas do SonarQube para avaliar o esforço, a principal métrica relacionada ao esforço é a métrica LOC. Isso se deve ao fato de que o esforço está diretamente relacionado à quantidade de linhas de código inseridas, independentemente da existência de novas classes, funções e arquivos. No entanto, é importante notar que as outras métricas também são significativas e complementares, fornecendo uma perspectiva sobre como os padrões arquiteturais se comportam em relação às unidades modulares, como funções, classes e arquivos. Logo, é essencial utilizar uma combinação dessas métricas em conjunto com o tempo despendido para se obter uma avaliação mais sólida.

[29]https://docs.sonarqube.org/latest/

4.2. Troca da biblioteca *typeorm* pelo *prisma*

Banco de dados relacionais ainda vigoram como o "padrão" para armazenamento de dados (NOLETO, 2021). Como linguagens de programação atuais usualmente adotam o paradigma de orientação a objetos (FEITOSA; COMARELLA, 2020), é comum os sistemas de software adotarem *frameworks* de ORM (FONSECA, 2020).

Nesse cenário, o sistema utilizado neste artigo adota a biblioteca *typeorm*. Porém, atualmente, a comunidade de desenvolvedores é muito ativa na biblioteca *open source prisma* cujo maior diferencial é a baixa complexidade de implementações comparada com as demais bibliotecas utilizadas no mercado (incluindo a *typeorm*), tornando o processo de desenvolvimento de software mais ágil (BUZZI, 2022) .

Esta seção, portanto, reporta a troca da biblioteca *typeorm* pelo *prisma* no sistema de software com arquitetura limpa e convencional.

4.2.1. Implementação Arquitetura Limpa

O processo de conversão da biblioteca *typeorm* para *prisma* pode ser descrito nas seguintes quatro etapas:

1. Inclusão e configuração da biblioteca *prisma*.

2. Transformação do esquema do banco de dados em um esquema do *prisma* por meio da execução do comando *prisma db pull*. O esquema gerado possibilita o autocompletar nos ambientes de desenvolvimento integrado, facilitando a manipulação de dados. A Figura 6 ilustra a transformação de uma tabela de agendamentos do banco de dados (à esquerda) em um trecho de código em um esquema do *prisma* (à direita).

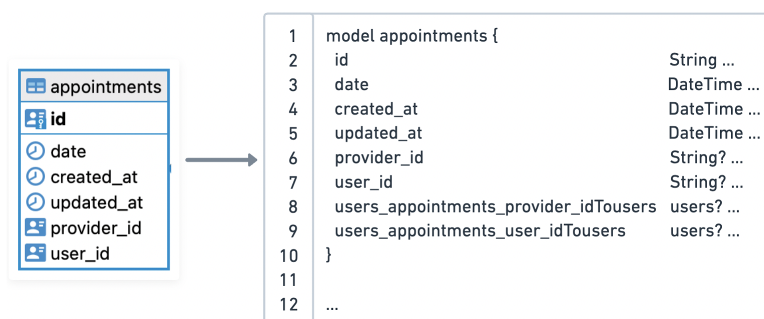

Figura 6. Mapeamento objeto-relacional da tabela de agendamentos do banco de dados para o esquema do *prisma*

3. Implementação (codificação) das operações de banco de dados utilizando a biblioteca *prisma*. Portanto, é preciso desenvolver uma classe que implementa a mesma interface utilizada nas implementações do *typeorm*, i.e., uma nova classe que cumpra com exigências do contrato que a interface previamente existente impõe.

4. Adequação das definições das fábricas de objeto de acesso à dados (repositórios). Em outras palavras, para de fato efetivar a troca da biblioteca ORM é preciso

alterar o arquivo de fábrica de cada repositório. Basicamente, cada fábrica contém a importação da classe de repositório que deve utilizar. Logo, a alteração de *typeorm* para *prisma* consiste no ajuste de uma única linha de código, substituindo a importação da classe de repositório relacionada à *typeorm* para aquela relacionada ao *prisma*. Para fins ilustrativos, os Códigos 4 e 5 representam a fábrica de repositório de agendamentos antes e depois da adequação das fábricas. É importante notar que apenas a linha 2 dos respectivos códigos são ajustadas.

```
1 import IAppointmentRepository from '@providers/
      repositories/i-appointment-repository';
2 import PostgreSQLAppointmentRepository from'
      @providers/repositories/typeorm/appointment-
      repository';
3
4 export const makeAppointmentRepository = ():
      IAppointmentRepository => {
5   return new PostgreSQLAppointmentRepository();
6 };
```

Código 4. Antes da troca de ORM no arquivo de fábrica do repositório de agendamentos no sistema com arquitetura limpa

```
1 import IAppointmentRepository from '@providers/
      repositories/i-appointment-repository';
2 import PostgreSQLAppointmentRepository from '
      @providers/repositories/prisma/appointment-
      repository';
3
4 export const makeAppointmentRepository = ():
      IAppointmentRepository => {
5   return new PostgreSQLAppointmentRepository();
6 };
```

Código 5. Após a troca de ORM no arquivo de fábrica do repositório de agendamentos no sistema com arquitetura limpa

4.2.2. Implementação Convencional

As etapas envolvidas na conversão da biblioteca *typeorm* para prisma podem ser divididas em quatro fases:

1. Assim como na implementação com arquitetura limpa da Seção 4.2.1, a instalação da biblioteca e configurações básicas para seu funcionamento.

2. Assim como na implementação com arquitetura limpa da Seção 4.2.1, a transformação do esquema do banco de dados em um esquema do *prisma*.

3. Detalhes de implementação das operações do *prisma* no banco de dados. Para atender às exigências da interface utilizada nas implementações do *typeorm*, é necessário criar uma nova classe que implemente essa mesma interface. O arquivo `appointment-repository.ts` fornece detalhes sobre a implementação do repositório de agendamentos. Ao observar o código, é notado que não há importações de entidades no arquivo. O tipo de dado que é trocado entre os arquivos é da tipagem do modelo de dado do *prisma*, o que implica em algumas outras modificações ao realizar a troca de biblioteca, em comparação com a troca realizada no sistema de software com arquitetura limpa.

4. Troca da biblioteca *typeorm* para o *prisma*. Para trocar a biblioteca ORM no sistema de software convencional, é necessário modificar o arquivo de fábrica que gerencia a injeção de dependência dos repositórios, utilizando a biblioteca *tsyringe*. Esse arquivo contém as importações das classes de repositórios que utilizam uma dada biblioteca ORM. Assim, a primeira etapa para a troca de *typeorm* para *prisma* envolve a modificação de uma única linha de código, removendo a importação da classe do repositório que utiliza o *typeorm* e adicionando a importação da classe do repositório que utiliza o *prisma*. Os Códigos 6 e 7 mostram a fábrica de repositórios antes e depois da troca de biblioteca.

```
...
import { container } from 'tsyringe';
import IAppointmentRepository from './
      IAppointmentRepository';
import AppointmentRepository from './typeorm/
      appointment-repository';
import IUserRepository from './IUserRepository'
      ;
import IUserTokenRepository from './
      IUserTokenRepository';
import UserRepository from './typeorm/user-
      repository';

container.registerSingleton<
      IAppointmentRepository>(
  'AppointmentRepository',
  AppointmentRepository,
);

container.registerSingleton<IUserRepository>('
      UserRepository', UserRepository);
...
```

Código 6. Antes da troca de ORM no arquivo de fábrica de repositórios no sistema convencional

```
...
import { container } from 'tsyringe';
import IAppointmentRepository from './
      IAppointmentRepository';
import AppointmentRepository from './prisma/
      appointment-repository';
import IUserRepository from './IUserRepository'
      ;
import IUserTokenRepository from './
      IUserTokenRepository';
import UserRepository from './prisma/user-
      repository';

container.registerSingleton<
      IAppointmentRepository>(
  'AppointmentRepository',
  AppointmentRepository,
);

container.registerSingleton<IUserRepository>('
      UserRepository', UserRepository);
...
```

Código 7. Após a troca de ORM no arquivo de fábrica de repositórios no sistema convencional

A mudança de biblioteca ORM é realizada nas linhas 4 e 7 dos Códigos 6 e 7, onde são importadas as classes dos repositórios. Isso requer a correção da interface que descreve o contrato dos repositórios, para que esses repositórios usem os modelos do *prisma* em vez do *typeorm*. A alteração é ilustrada nos Códigos 8 e 9.

```
...
import Appointment from '@models/Appointment';

export default interface IAppointmentRepository
    {
  save(data: Appointment): Promise<
      Appointment>;
  findByDate(date: Date, providerId: string):
      Promise<Appointment | null>;
  ...
}
```

Código 8. Antes da troca de ORM no arquivo que contém a interface do repositório no sistema convencional

```
...
import { appointments } from '@prisma/client';

export default interface IAppointmentRepository
    {
  save(data: appointments): Promise<
      appointments>;
  findByDate(date: Date, providerId: string):
      Promise<appointments | null>;
  ...
}
```

Código 9. Após a troca de ORM no arquivo que contém a interface do repositório no sistema convencional

Ao analisar o código, é notado que a tipagem do parâmetro de entrada no método *save* e a tipagem de retorno agora são do tipo do modelo do *prisma*. Além disso, as classes dos repositórios utilizadas nos testes de unidade também precisaram se adaptar ao novo contrato, o que incluiu alterações nas tipagens. Da mesma forma, todas as classes dos arquivos que representam os casos de uso precisaram se adaptar, removendo importações e tipagens usadas pelo *typeorm* e substituindo-as pelas importações e tipagens do *prisma*.

4.2.3. Análise Quantitativa

A metodologia previamente definida na Seção 4.1 estabelece a aferição de várias métricas e uma análise do esforço em termos temporais de cada troca de biblioteca em cada um dos sistemas (arquitetura limpa e convencional).

Métricas: As métricas auferidas na Tabela 2 indicam – de forma sumarizada – que a quantidade de artefatos de código fonte necessários para a biblioteca *prisma* é muito inferior ao necessário para a *typeorm*. Por exemplo, para qualquer métrica, o valor diminuiu após a troca para as duas implementações. Portanto, um custo menor de implementação pode indicar que a biblioteca *prisma* é mais simples de utilizar e necessita de menos código para alcançar as mesmas funcionalidades da biblioteca *typeorm*.

Já ao comparar as implementações, fica evidenciado que o sistema convencional apresentou menor redução de artefatos de código fonte. Segundo Gamma et al. (1995), a separação de responsabilidade e o uso de padrões de projeto são fundamentais para desenvolver sistemas escaláveis e flexíveis. Desse modo, infere-se que a falta de separação de responsabilidades e acoplamentos fortes no sistema convencional podem dificultar a substituição de bibliotecas, resultando na escrita de código adicional para adaptá-las ao sistema existente.

Tabela 2. Métricas auferidas na troca da biblioteca de ORM

Métrica	***Arquitetura Limpa***			**Convencional**		
	Antes	*Depois*	*Diferença*	*Antes*	*Depois*	*Diferença*
Linhas de código	2.988	2.527	-461	2.104	1.773	-331
Número de Funções	241	204	-37	131	110	-21
Número de Classes	70	53	-17	48	38	-10
Número de Arquivos	138	118	-20	92	83	-9

Esforço: Por um lado, os tempos despendidos em cada uma das etapas reportados na Tabela 3 confirmam uma maior complexidade arquitetural na arquitetura limpa cujo desenvolvimento de fato se tornou mais lento (115 minutos vs. 105 minutos da implementação convencional). Por outro lado, o tempo da efetiva troca de biblioteca na implementação com arquitetura limpa é bem menor (5 minutos vs. 40 minutos da implementação convencional). Isso evidencia que a arquitetura limpa tem um acoplamento menor, o que torna esse processo de troca de biblioteca mais ágil.

Tabela 3. Tempos despendidos (por etapa) na troca da biblioteca de ORM

Etapa	***Arquitetura Limpa***	**Convencional**
Estudo do *prisma*	120 minutos	120 minutos
Desenvolvimento usando *prisma*	115 minutos	105 minutos
Troca efetiva da biblioteca	5 minutos	40 minutos
Testes	30 minutos	30 minutos
Total	**270 minutos**	**295 minutos**

4.3. Troca da biblioteca *express* pelo *apollo-server*

Embora REST seja a abordagem mais popular e mais amplamente utilizada no mercado para desenvolvimento de APIs, essa estratégia contém pontos negativos, por exemplo, *over-fetching* (excesso de dados) e *under-fetching* (dados insuficientes) (FORTE, 2022). Diante disso, o Facebook lançou *GraphQL*[30] como uma nova abordagem

[30]https://graphql.org

de desenvolvimento de APIs que oferece eficiência e flexibilidade na integração de APIs (VIDANYA, 2021).

O objetivo principal consiste em tornar a API mais voltada ao cliente por usar busca de dados declarativa nas requisições através de um servidor *GraphQL*, eliminando problemas de *over-fetching* e *under-fetching* (CLARK, 2022).

O sistema utilizado neste artigo adota a biblioteca *express* como estratégia de desenvolvimento de API REST. Já a abordagem de desenvolvimento de API *GraphQL* necessita de uma biblioteca que sirva como servidor *GraphQL*. Nesse cenário, a biblioteca *apollo-server* é uma das implementações mais conhecidas do *GraphQL*.

Esta seção, portanto, reporta a troca da biblioteca *express* pelo *apollo-server* no sistema de software com arquitetura limpa e convencional.

4.3.1. Implementação Arquitetura Limpa

No início do processo de substituição da biblioteca *express* pela biblioteca *prisma*, o primeiro autor desta pesquisa identificou algumas incongruências no projeto que não estavam de acordo com a arquitetura limpa. O primeiro autor deste estudo verificou que os controladores estavam fazendo uso direto da biblioteca externa *express* e que o código do adaptador das requisições não era genérico o suficiente para atender tanto ao servidor REST quanto ao servidor *GraphQL*. Dessa forma, antes de prosseguir com a substituição das bibliotecas, o primeiro autor realizou as devidas correções.

Em seguida, o processo de conversão da biblioteca *express* para *apollo-server* pode ser descrito nas seguintes quatro etapas:

1. Inclusão e configuração da biblioteca *apollo-server*.

2. Criação de diretivas dos *middlewares* de autenticação e taxa de limitação de requisições. Nesse caso, foram utilizadas outras bibliotecas para lidar com as nuances do servidor *GraphQL*: *@graphql-tools/utils*[31], *graphql-rate-limit-directive*[32] e *graphql*.

3. Criação das estruturas do *GraphQL* para cada caso de uso do sistema de software.

4. Troca do servidor REST pelo servidor *GraphQL*. Para fins ilustrativos, os Códigos 10 e 11 representam o arquivo que aplica o servidor antes e depois da adequação. É importante notar que só a linha 3 dos respectivos códigos são ajustadas.

```
1 import setupExpressServer from './express';
2
3 const app = setupExpressServer();
4
5 export default app;
```

Código 10. Antes da troca do express pelo apollo-server no arquivo que aplica o servidor no sistema com arquitetura limpa

```
1 import setupApolloServer from './apollo-server';
2
3 const app = setupApolloServer();
4
5 export default app;
```

Código 11. Após a troca do express pelo apollo-server no arquivo que aplica o servidor no sistema com arquitetura limpa

[31] https://www.npmjs.com/package/@graphql-tools/utils
[32] https://www.npmjs.com/package/graphql-rate-limit-directive

4.3.2. Implementação Convencional

O processo de conversão da biblioteca *express* para *apollo-server* pode ser descrito nas seguintes cinco etapas:

1. Assim como na implementação com arquitetura limpa da Seção 4.3.1, inclusão e configuração da biblioteca *apollo-server*.

2. Criação do adaptador para servidor *GraphQL*, conforme pode ser observado no Código 12. É importante notar, nas linhas 8 e 9, o uso de *args* e *context* para obter os dados da requisição, diferente de um servidor REST em que os parâmetros da requisição são acessados via *query*, *route* ou *body params*.

```
...
export const adaptResolver = async (
  controller: any,
  args: any,
  context?: any,
): Promise<any> => {
  const request = {
    ...(args || {}),
    userId: context?.req?.userId,
  };

  const httpResponse = await controller.handle(request);
  ...
}
```

Código 12. **Adaptador para servidor GraphQL**

3. Assim como na implementação com arquitetura limpa da Seção 4.3.1, criação de diretivas dos *middlewares* de autenticação e taxa de limitação de requisições.

4. Assim como na implementação com arquitetura limpa da Seção 4.3.1, criação das estruturas do *GraphQL*.

5. Assim como na implementação com arquitetura limpa da Seção 4.3.1, troca do servidor REST pelo servidor *GraphQL*.

6. Adequação dos arquivos que implementam os controladores ao adaptador do servidor *GraphQL*. No cenário do servidor REST, os controladores recebem como parâmetro um objeto que contém propriedades e valores dependendo da forma que a requisição foi feita, por exemplo, usando *query*, *route* ou *body params*. Já no servidor *GraphQL*, não existe essa distinção e, portanto, é preciso adequar os arquivos que implementam os controladores ao novo formato de dados.

4.3.3. Análise Quantitativa

A metodologia descrita na Seção 4.1 inclui a medição de várias métricas e a avaliação do tempo necessário para troca de biblioteca em ambos sistemas (arquitetura limpa e convencional).

Métricas: Os dados apresentados na Tabela 4 mostram que a biblioteca *apollo-server* requer mais artefatos de código fonte do que a biblioteca *express*. Com exceção da quantidade de classes, que permaneceu constante nos dois sistemas, é possível observar o

aumento nos valores das outras métricas após a troca de biblioteca para ambas as implementações. Com isso, é possível inferir que a biblioteca *apollo-server* é mais complexa de se utilizar e requer mais código para alcançar as mesmas funcionalidades que a *express*.

Tabela 4. Métricas auferidas na troca da biblioteca de abordagem de desenvolvimento de API

Métrica	*Arquitetura Limpa*			Convencional		
	Antes	*Depois*	*Diferença*	*Antes*	*Depois*	*Diferença*
Linhas de código	2.988	3.039	51	2.104	2.243	139
Número de Funções	241	247	6	131	153	22
Número de Classes	70	70	0	48	48	0
Número de Arquivos	138	143	5	92	103	11

Já ao analisar as implementações, foi observado que a troca de biblioteca no sistema convencional resultou em um aumento significativo na quantidade de artefatos de código fonte em comparação ao sistema com arquitetura limpa. De acordo com Martin (2002), é fundamental para o desenvolvimento de sistemas escaláveis e flexíveis a separação de responsabilidade e o uso de padrões de projeto. Portanto, infere-se que a falta de separação de responsabilidades e acoplamentos fortes em um sistema convencional pode dificultar a substituição de bibliotecas, o que pode exigir o desenvolvimento de código adicional para adaptar a nova biblioteca ao sistema existente.

Esforço: Por outra perspectiva, a Tabela 5 apresenta uma análise dos tempos despendidos em cada etapa. É possível notar que a implementação com arquitetura limpa apresentou uma maior complexidade arquitetural, com um tempo de desenvolvimento mais longo (435 minutos) em comparação com a implementação convencional (420 minutos). Por outro lado, a implementação com arquitetura limpa apresentou uma vantagem significativa na troca de biblioteca, com um tempo menor (5 minutos) em comparação com a implementação convencional (25 minutos). Portanto, os dados coletados sugerem que a arquitetura limpa apresenta menor grau de acoplamento, o que resulta em uma troca de biblioteca mais eficiente.

Tabela 5. Tempos despendidos (por etapa) na troca da biblioteca de abordagem de desenvolvimento de API

Etapa	*Arquitetura Limpa*	Convencional
Estudo do *apollo-server*	600 minutos	600 minutos
Desenvolvimento usando *apollo-server*	435 minutos	420 minutos
Troca efetiva da biblioteca	5 minutos	25 minutos
Testes	30 minutos	30 minutos
Total	**1.070 minutos**	**1.075 minutos**

4.4. Troca da biblioteca *bull* pelo *bullmq*

Fila de mensagens permite que servidores trabalhem de forma assíncrona de modo a continuarem operando sem interrupções (YANG, 2020). Como exemplo, considere a tarefa de envio de e-mail em que o servidor **A** envia os dados para o servidor **B** realizar o

disparo de e-mail. Enquanto o servidor de destino estiver ocupado, o servidor remetente continuará operando sem paralisações.

Bull é uma biblioteca que proporciona uma implementação sólida e de alta velocidade de um sistema de filas (EKUMA, 2020). Já a *bullmq* é uma biblioteca mais moderna, baseada na biblioteca *bull*, que oferece recursos e funcionalidades extras para gerenciar filas de mensagens em larga escala (PANDIYAN, 2023).

Esta seção, portanto, reporta a troca da biblioteca *bull* pelo *bullmq* no sistema de software com arquitetura limpa e convencional.

4.4.1. Implementação Arquitetura Limpa

O processo de conversão da biblioteca *bull* pela *bullmq* pode ser descrito nas seguintes três etapas:

1. Inclusão da biblioteca *bullmq*.

2. Implementação (codificaçao) das operações da fila utilizando a biblioteca *bullmq*. Portanto, é preciso desenvolver uma classe que implementa a mesma interface utilizada nas implementações do *bull*, i.e., uma nova classe que cumpra com as exigências do contrato que a interface previamente existente impõe.

3. Adequação das definições das fábricas do *provider* da fila de mensagens. Em outras palavras, para de fato efetivar a troca da biblioteca de fila de mensagens é preciso alterar o arquivo de fábrica do *provider*. Basicamente, cada fábrica contém a importação da classe de repositório que deve utilizar. Logo, a alteração de *bull* para *bullmq* consiste no ajuste de uma única linha de código, substituindo a importação da classe de repositório relacionada à *bull* para aquela relacionada à *bullmq*. Para fins ilustrativos, os Códigos 13 e 14 representam a fábrica do *provider* de fila antes e depois da adequação das fábricas. É importante notar que apenas as linhas 1 e 5 dos respectivos códigos são ajustadas.

```
1 import { BullProvider } from '@providers/queue/
      implementations/bull.provider';
2 import { IQueueProvider } from '@providers/
      queue/queue-provider.interface';
3
4 export const makeQueueProvider = ():
      IQueueProvider => {
5   return new BullProvider();
6 };
```

Código 13. Antes da troca do *bull* pelo *bullmq* no arquivo de fábrica do provider de fila no sistema com arquitetura limpa

```
1 import { BullmqProvider } from '@providers/
      queue/implementations/bullmq.provider';
2 import { IQueueProvider } from '@providers/
      queue/queue-provider.interface';
3
4 export const makeQueueProivder = ():
      IQueueProvider => {
5   return new BullmqProvider();
6 };
```

Código 14. Após a troca do *bull* pelo *bullmq* no arquivo de fábrica do provider de fila no sistema com arquitetura limpa

4.4.2. Implementação Convencional

As etapas envolvidas na conversão da biblioteca *bull* pela *bullmq* podem ser divididas em três fases:

1. Assim como na implementação com arquitetura limpa da Seção 4.4.1, a inclusão da biblioteca *bullmq*.
2. Na implementação com arquitetura limpa da Seção 4.4.1, a implementação das operações de fila usando a biblioteca *bullmq*.
3. Troca da biblioteca *bull* para o *bullmq*. Para trocar a biblioteca de fila no sistema de software convencional é necessário modificar o arquivo de fábrica que gerencia a injeção de dependência dos *providers*, utilizando a biblioteca *tsyringe*. Esse arquivo de fábrica contém as importações das classes de *providers* que utilizam uma determinada biblioteca de fila. Assim, a primeira etapa para a troca de *bull* para *bullmq* envolve a modificação de apenas duas linhas de código. Os Códigos 15 e 16 mostram a fábrica de repositórios antes e depois da troca de biblioteca.

```
1 import { container } from 'tsyringe';
2 import BullProvider from './implementations/
      bull.provider';
3 import IQueueProvider from './queue-provider.
      interface';
4
5 container.registerSingleton<IQueueProvider>('
      QueueProvider', BullProvider);
```

Código 15. Antes da troca do *bull* pelo *bullmq* no arquivo de fábrica do provider de fila no sistema convencional

```
1 import { container } from 'tsyringe';
2 import BullMqProvider from './implementations/
      bullmq.provider';
3 import IQueueProvider from './queue-provider.
      interface';
4
5 container.registerSingleton<IQueueProvider>('
      QueueProvider', BullMqProvider);
```

Código 16. Após a troca do *bull* pelo *bullmq* no arquivo de fábrica do provider de fila no sistema convencional

4.4.3. Análise Quantitativa

A metodologia descrita na Seção 4.1 foi utilizada para avaliar várias métricas e analisar o esforço temporal necessário para a troca de biblioteca em ambos os sistemas (arquitetura limpa e convencional).

Métricas: De acordo com os dados apresentados na Tabela 6, é possível observar que a biblioteca *bullmq* requer um número maior de artefatos de código fonte em comparação com a biblioteca *bull*. Com exceção da quantidade de classes, que permaneceu inalterada nos dois sistemas, é possível constatar o aumento nas outras métricas após a substituição da biblioteca para as duas implementações. Com base nesses resultados, é possível inferir que a biblioteca *bullmq* é mais complexa de ser utilizada e requer mais código para alcançar as mesmas funcionalidades que a biblioteca anterior.

Tabela 6. Métricas auferidas na troca da biblioteca de fila

Métrica	***Arquitetura Limpa***			**Convencional**		
	Antes	*Depois*	*Diferença*	*Antes*	*Depois*	*Diferença*
Linhas de código	3.098	3.128	30	2.220	2.250	30
Número de Funções	256	257	1	147	148	1
Número de Classes	75	75	0	53	53	0
Número de Arquivos	144	144	0	100	100	0

Já ao analisar as implementações, é possível constatar que a troca de biblioteca em ambos sistemas não teve diferença na quantidade de artefatos de código fonte. Isso indica

que a arquitetura de ambos os sistemas é flexível o suficiente para adaptar-se à troca da biblioteca sem afetar significativamente a estrutura do código fonte.

Esforço: Por outro lado, as resultados apresentados na Tabela 7 indicam que, embora a implementação com arquitetura limpa tenha requerido um tempo de desenvolvimento maior (425 minutos) do que a convencional (420 minutos), ela ofereceu uma vantagem significativa na troca de biblioteca. Enquanto a implementação com arquitetura limpa levou apenas 5 minutos para realizar essa tarefa, a convencional precisou de 15 minutos. Isso sugere que a arquitetura limpa tem um menor grau de acoplamento, o que resulta em uma troca de biblioteca mais eficiente.

Tabela 7. Tempos despendidos (por etapa) na troca da biblioteca de fila

Etapa	***Arquitetura Limpa***	**Convencional**
Estudo do *bullmq*	600 minutos	600 minutos
Desenvolvimento usando *bullmq*	425 minutos	420 minutos
Troca efetiva da biblioteca	5 minutos	15 minutos
Testes	30 minutos	30 minutos
Total	**1.060 minutos**	**1.065 minutos**

4.5. Análise e Discussão

A análise dos dados apresentados nesta seção permitiu concluir que a biblioteca *prisma* apresentou uma menor quantidade de artefatos de código fonte em ambos os sistemas (convencional e com arquitetura limpa) quando comparada ao *typeorm*. Quando comparadas as bibliotecas *express* e *bull* com as bibliotecas *apollo-server* e *bullmq*, respectivamente, observou-se que a *apollo-server* e a *bullmq* apresentam uma maior quantidade de artefatos de código fonte. De acordo com McConnell (2004), a complexidade de um sistema aumenta com o número de linhas de código, pois aumenta o número de partes do sistema que precisam ser entendidas e mantidas. Desse modo, infere-se que as bibliotecas *typeorm*, *apollo-server* e *bullmq* agregam mais complexidade ao sistema em comparação com as bibliotecas *prisma*, *express* e *bull*, respectivamente.

Ademais, foi observado que a troca da biblioteca *prisma* gerou menos esforço no sistema com arquitetura limpa ao apresentar uma diminuição significativa na quantidade de linhas de código (-461 LOC vs. -331 LOC da implementação convencional). Da mesma forma, a troca da biblioteca *apollo-server* gerou menos esforço no sistema com arquitetura limpa, porém com um aumento expressivo na quantidade de linhas de código na implementação convencional (139 LOC vs. 51 LOC da implementação com arquitetura limpa). Por outro lado, a troca da biblioteca *bullmq* demostrou esforços equivalentes em ambos os sistemas, visto que a quantidade de linhas de código aumentaram igualmente. De acordo com Taylor, Medvidovic e Dashofy (1999), é crucial projetar sistemas que sejam flexíveis e adaptáveis para lidar com mudanças. Seguindo essa linha de raciocínio, infere-se que o sistema com arquitetura limpa é mais flexível e adaptável que o sistema convencional, pois foi mais receptível às mudanças após a troca das bibliotecas *prisma* e *apollo-server*.

Verificou-se também que a implementação das bibliotecas no sistema com arquitetura limpa requereu mais esforço ao apresentar um tempo maior (*prisma*: 115 minutos

vs. 105 minutos da implementação convencional, *apollo-server*: 435 minutos vs. 420 minutos da implementação convencional e *bullmq*: 425 minutos vs. 420 minutos da implementação convencional). Contudo, a troca das bibliotecas no sistema com arquitetura limpa gerou um esforço menor ao demostrar um tempo inferior (*prisma*: 5 minutos vs. 40 minutos da implementação convencional, *apollo-server*: 5 minutos vs. 25 minutos da implementação convencional e *bullmq*: 5 minutos vs. 15 minutos da implementação convencional). De acordo com Feathers (2004), o acoplamento excessivo entre módulos pode afetar negativamente a manutenibilidade e evolução do sistema de software. Desse modo, evidenciou-se que o sistema com arquitetura limpa tem um acoplamento menor que o sistema convencional, devido ao fato de que as trocas de bibliotecas são mais rápidas, tornando-o mais fácil de manter.

É importante salientar que a arquitetura do sistema convencional empregado neste estudo é acima do que se considera razoável. Se a arquitetura de tal sistema fosse mais ordinária, os resultados poderiam ser mais expressivos. Portanto, seria benéfico conduzir o experimento em um sistema com uma arquitetura mais rudimentar como tema de estudo futuro.

5. Ameaças à validade

Pelo menos três ameaças a validade devem ser contempladas.

Dataset: A criação do *dataset* foi efetuada pelo primeiro autor deste trabalho, por meio da conversão de um sistema convencional. Embora algumas pessoas possam questionar tal transformação, o primeiro autor desta pesquisa possui extensa especialização nas tecnologias empregadas. Além disso, o processo de conversão é descrito na Seção 3.3 e o projeto original do sistema convencional foi elaborado por terceiros.

Troca de bibliotecas: Algumas pessoas podem argumentar que a primeira substituição de biblioteca, seja no sistema convencional ou na arquitetura limpa, sempre será mais lenta. Para evitar esse viés, é realizada a substituição na arquitetura convencional após a substituição na arquitetura limpa. Ademais, algumas pessoas poderiam alegar questões de cansaço e fadiga. Por conseguinte, a Seção 5.1 descreve uma metodologia que adota sessões consecutivas de uma hora por dia para a substituição de bibliotecas, a fim de evitar possíveis erros de medição devido aos fatores supracitados.

Mensuração: Algumas pessoas poderiam questionar que todo processo de mensuração foi realizado por apenas uma pessoa. Essa é uma limitação deste estudo, no entanto, o escopo de um TCC não exige uma experimentação mais elaborada, embora isso seja importante e seja considerado como um trabalho futuro.

6. Conclusão

A arquitetura limpa é um modelo arquitetural que busca tornar o código fácil de entender, manter e evoluir, através da separação das responsabilidades em camadas independentes para melhorar testabilidade, escalabilidade e manutenibilidade (MARTIN et al., 2018). No entanto, a arquitetura limpa aumenta a verbosidade e a complexidade do sistema devido ao demasiado uso de interfaces e classes adicionais para deixar o código mais desacoplado e coeso, além de demandar uma curva de aprendizado maior, especi-

almente para desenvolvedores acostumados com padrões menos complexos (MASOTTI, 2021).

Diante desse cenário, este trabalho de conclusão de curso avalia o esforço de trocas de bibliotecas em sistemas com e sem arquitetura limpa. Basicamente, um mesmo sistema desenvolvido em uma arquitetura convencional (BRAZ, 2021) foi convertido usando todas as diretrizes e práticas da arquitetura limpa pelo primeiro autor deste artigo. Visando fazer uma análise de esforço para adaptar um sistema de software convencional em sistema com arquitetura limpa, a metodologia consistiu primeiramente fazer um levantamento das decisões de projeto que não vão ao encontro do estabelecido pela arquitetura limpa. Consequentemente, as decisões de projeto foram corrigidas e simultaneamente foram realizadas implementações para adequação à arquitetura limpa, seguida de uma análise empírica dos esforços dos ajustes necessários para tal conversão. Portanto, essa é a primeira contribuição do artigo, além de prover um *dataset* onde têm dois sistemas de software equivalentes, um com arquitetura convencional e outro seguindo todos os princípios da arquitetura limpa.

Em seguida, procedeu-se com a substituição de bibliotecas distintas utilizadas, tanto no sistema convencional quanto no sistema com arquitetura limpa. A biblioteca *typeorm* foi trocada pela *prisma*, a *express* pelo *apollo-server* e a *bull* pela *bullmq*. Como resultado, foi observado que a troca da biblioteca *prisma* gerou menos esforço no sistema com arquitetura limpa ao apresentar uma diminuição significativa na quantidade de linhas de código (-461 LOC vs. -331 LOC da implementação convencional). Da mesma forma, a troca da biblioteca *apollo-server* gerou menos esforço no sistema com arquitetura limpa, porém com um aumento expressivo na quantidade de linhas de código na implementação convencional (139 LOC vs. 51 LOC da implementação com arquitetura limpa). Por outro lado, a troca da biblioteca *bullmq* demostrou esforços equivalentes em ambos os sistemas, visto que a quantidade de linhas de código aumentaram igualmente.

Verificou-se também que a implementação das bibliotecas no sistema com arquitetura limpa requereu mais esforço ao apresentar um tempo maior (*prisma*: 115 minutos vs. 105 minutos da implementação convencional, *apollo-server*: 435 minutos vs. 420 minutos da implementação convencional e *bullmq*: 425 minutos vs. 420 minutos da implementação convencional). Contudo, a troca das bibliotecas no sistema com arquitetura limpa gerou um esforço menor ao demostrar um tempo inferior (*prisma*: 5 minutos vs. 40 minutos da implementação convencional, *apollo-server*: 5 minutos vs. 25 minutos da implementação convencional e *bullmq*: 5 minutos vs. 15 minutos da implementação convencional). Desse modo, conclui-se que, embora o sistema com arquitetura limpa requisitou um tempo maior de implementação das bibliotecas, demonstrou-se um sistema com um acoplamento menor que o sistema convencional, devido ao fato de que as trocas de bibliotecas são mais rápidas, tornando-o mais fácil de manter.

Diante disso, os seguintes trabalhos futuros podem ser vislumbrados: (i) analisar a troca de outros tipos de bibliotecas em sistemas com e sem arquitetura limpa com objetivo de identificar em quais dos sistemas as substituições de bibliotecas resultam em mais, menos ou iguais esforços em termos de quantidade de linhas de código, visto que neste estudo foi constatado que as trocas forneceram resultados diferentes em relação ao LOC; (ii) incluir a participação de mais pessoas no processo de mensuração, de preferência usando o mesmo *dataset* provido neste artigo, com o objetivo de reforçar a confiabilidade

dos resultados; (iii) aplicar outras metodologias para ajudar o desenvolvedor na escolha de bibliotecas apropriadas para o seu projeto. A metodologia pode incluir análise de outras métricas, avaliação de desempenho, *feedback* de usuários e comparação com outras opções semelhantes; e (iv) reconduzir o mesmo experimento comparando com um sistema com uma arquitetura mais rudimentar para avaliar se os resultados obtidos seriam mais significativos.

Referências

BARRO, B. B. **O que são Frameworks e quais os mais utilizados**. 2022. Disponível em: <https://www.hostinger.com.br/tutoriais/frameworks>. Acesso em: 13 fev. 2023.

BRAZ, L. H. D. **Domain Driven Design: Vantagens e desvantagens em seu uso**. Trabalho de conclusão de curso de Ciência da Computação. Universidade Federal de Lavras, 2021. 1-52 p.

BUZZI, F. **Prisma: uma das melhores coisas que já aconteceu no ecossistema?** 2022. Disponível em: <https://blog.rocketseat.com.br/prisma-uma-das-melhores-coisa-que-ja-aconteceu-no-ecossistema/>. Acesso em: 13 fev. 2023.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. **IEEE Transactions on Software Engineering**, v. 20, n. 6, p. 476–493, 1994.

CLARK, M. **GraphQL vs. REST | Quais são as diferenças?** 2022. Disponível em: <https://blog.back4app.com/pt/graphql-vs-rest-quais-sao-as-diferencas/#O_que_e_GraphQL>. Acesso em: 13 fev. 2023.

COCKBURN, A. **Hexagonal architecture**. 2005. Disponível em: <https://alistair.cockburn.us/hexagonal-architecture>. Acesso em: 13 fev. 2023.

EKUMA, G. **Asynchronous task processing in Node.js with Bull**. 2020. Disponível em: <https://blog.logrocket.com/asynchronous-task-processing-in-node-js-with-bull/>. Acesso em: 13 fev. 2023.

EVANS, E.; EVANS, E. J. **Domain-driven design: tackling complexity in the heart of software**. [S.l.]: Addison-Wesley Professional, 2004.

FEATHERS, M. **Working effectively with legacy code**. [S.l.]: Pearson Education, 2004.

FEITOSA, S. da S.; COMARELLA, R. L. Aprendendo conceitos de orientação a objetos usando as ferramentas Scratch e Snap! **Anais do Computer on the Beach**, v. 11, n. 1, p. 490–496, 2020.

FENTON, N. E. **Software Metrics: A Rigorous and Practical Approach**. [S.l.]: CRC Press, 1999.

FIELDING, R. T. **Architectural styles and the design of network-based software architectures**. [S.l.]: Ph.D. thesis. University of California, 2000.

FILHO, J. **Trabalhando com UUID no Laravel**. 2021. Disponível em: <https://laraveling.tech/trabalhando-com-uuid-no-laravel/>. Acesso em: 13 fev. 2023.

FONSECA, E. **O que é ORM?** 2020. Disponível em: <https://www.treinaweb.com.br/blog/o-que-e-orm>. Acesso em: 13 fev. 2023.

FORTE, G. **Over-Fetching and Under-Fetching: REST APIs' Exhaustion Signs**. 2022. Disponível em: <https://www.programmersinc.com/over-fetching-and-under-fetching-rest-apis-exhaustion-signs/>. Acesso em: 13 fev. 2023.

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. [S.l.]: Addison-Wesley, 1995.

HENRY, S.; KAFURA, D. Software metrics: a research evaluation. **IEEE Transactions on Software Engineering**, SE-7, n. 6, p. 576–590, 1981.

IEEE. **IEEE Recommended Practice for Architecture Description of Software- Intensive Systems, ANSI/IEEE Std 1471, ISO/IEC 42010**. 2000. Disponível em: <http://www.iso-architecture.org/ieee-1471/>. Acesso em: 13 fev. 2023.

JACOBSON, I. **Object-oriented software engineering: A use case driven approach**. [S.l.]: Addison Wesley, 2004.

LEMOS, O. L. A. **Arquitetura Limpa na Prática**. [S.l.]: Auto-publicado em: <http://www.otaviolemos.com.br>, 2021.

MARTIN, R. C. **Agile Software Development, Principles, Patterns, and Practices**. [S.l.]: Prentice Hall, 2002.

MARTIN, R. C. et al. **Clean architecture: a craftsman's guide to software structure and design**. [S.l.]: Prentice Hall, 2018.

MASOTTI, D. **Clean Architecture: descubra o que é e onde aplicar Arquitetura Limpa**. 2021. Disponível em: <https://www.zup.com.br/blog/clean-architecture-arquitetura-limpa>. Acesso em: 13 jan. 2022.

MCCONNELL, S. **Code Complete: A Practical Handbook of Software Construction**. [S.l.]: Microsoft Press, 2004.

NOLETO, C. **Bancos de Dados**. 2021. Disponível em: <https://blog.betrybe.com/tecnologia/bancos-de-dados/>. Acesso em: 13 fev. 2023.

OKTAFIANI, I.; HENDRADJAYA, B. Software metrics proposal for conformity checking of class diagram to solid design principles. In: **5th International Conference on Data and Software Engineering (ICoDSE)**. [S.l.: s.n.], 2018. p. 1–6.

OLIVEIRA, J. **Log4j: entenda mais sobre a vulnerabilidade do bug**. 2021. Disponível em: <https://www.alura.com.br/artigos/log4j-entenda-sobre-vulnerabilidade>. Acesso em: 13 fev. 2023.

PANDIYAN, M. **Bullmq — Message Queue in Node.js**. 2023. Disponível em: <https://medium.com/@marudhupandiyang/bullmq-message-queue-in-node-js-bca24bac7f8a>. Acesso em: 13 fev. 2023.

REENSKAUG, T. **The common sense of object oriented programming**. 2009. Disponível em: <http://folk.uio.no/trygver/2009/commonsense.pdf>. Acesso em: 13 fev. 2023.

TAYLOR, R. N.; MEDVIDOVIC, N.; DASHOFY, E. M. **Software Architecture: Foundations, Theory, and Practice**. [S.l.]: John Wiley & Sons, 2019.

TRIPATHY, P.; NAIK, K. **Software evolution and maintenance: a practitioner's approach**. [S.l.]: John Wiley & Sons, 2014.

VALENTE, M. T. **Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade**. 2020. Disponível em: <https://engsoftmoderna.info>. Acesso em: 13 fev. 2023.

VIDANYA, B. **GraphQL vs. REST: Qual o melhor para o desenvolvimento de API?** 2021. Disponível em: <https://www.hostinger.com.br/blog/graphql-vs-rest-qual-o-melhor-para-o-desenvolvimento-de-api/>. Acesso em: 13 fev. 2023.

YANG, L. . P. **System Design - Message Queues**. 2020. Disponível em: <https://medium.com/must-know-computer-science/system-design-message-queues-245612428a22>. Acesso em: 13 fev. 2023.