Troca de Bibliotecas em Sistemas com e sem Arquitetura Limpa: Uma Análise de Esforço

Vinicius T. Pimenta[1], **Elder Cirilo**[2], **Ricardo Terra**[1]

[1] Departamento de Ciência da Computação
Universidade Federal de Lavras (UFLA)
Lavras – MG – Brasil

[2]Departamento de Ciência da Computação
Universidade Federal de São João del-Rei (UFSJ)
São João del-Rei – MG – Brasil

viniciustavarespimenta@gmail.com, elder@ufsj.edu.br, terra@ufla.br

Resumo. *A arquitetura limpa é 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. No entanto, essa arquitetura aumenta a verbosidade e a complexidade do sistema devido ao uso de muitas interfaces e classes adicionais, exigindo uma maior curva de aprendizado. Este trabalho avalia o esforço de troca de bibliotecas em sistemas com e sem arquitetura limpa. Um sistema desenvolvido com uma arquitetura convencional foi convertido para seguir todas as diretrizes da arquitetura limpa. Em seguida, três bibliotecas foram substituídas: typeorm por prisma, express por apollo-server e bull por bullmq. O esforço foi medido em termos de linhas de código alteradas e tempo necessário para as trocas. A conclusão foi que a arquitetura limpa apresentou menor esforço para trocar as bibliotecas, mas maior tempo de implementação, resultando em um sistema mais fácil de manter.*

1. Introdução

A arquitetura limpa (a.k.a., clean architecture) 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 [12].

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 [13]. Diante desse cenário, este trabalho avalia o esforço de troca de bibliotecas em sistemas com e sem arquitetura limpa, convertendo um sistema desenvolvido inicialmente de forma convencional. As bibliotecas *typeorm*[1], *express*[2] e *bull*[3] foram substituídas por *prisma*[4], *apollo-server*[5]

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

e *bullmq*[6], respectivamente. 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 resultados, o estudo conclui 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 define o sistema alvo utilizado e relata a conversão do sistema convencional em arquitetura limpa. A Seção 3 descreve a metodologia para realização das trocas bibliotecas, efetua as substituições e apresenta as análises e as discussões de cada substituição. A Seção 4 descreve as ameaças à validade do estudo. Por fim, a Seção 5 conclui.

2. Sistema

Adotou-se um sistema de barbearias desenvolvido por terceiros [1]. Essa aplicação é projetada para auxiliar no gerenciamento de barbearias, permitindo a administração do fluxo de agendamento, solicitações e atendimentos dos serviços prestados [1]. Para realização dos estudos, foi utilizada a implementação SEM DDD (chamada Implementação Convencional nesse artigo). Realizou-se uma modificação dessa implementação para um sistema que atende as características e conformidades arquiteturais da arquitetura limpa.

Para fins de análise, comparação e/ou replicabilidade do estudo, é importante mencionar que o código completo dos projetos e das conversões está disponível em:

https://anonymous.4open.science/r/vem-2024

2.1. Metodologia

Realizou-se o seguinte conjunto de passos:

1. Utilização de um projeto de terceiros [1];
2. Estudo do código e criação de um diagrama arquitetural usando engenharia reversa, modelando os elementos da arquitetura convencional conforme as camadas da arquitetura limpa e identificando decisões desalinhadas com seus princípios;
3. Modificações no projeto para corrigir essas decisões e adequá-lo à arquitetura limpa, seguindo diretrizes e boas práticas; e
4. Análise empírica dos esforços necessários para os ajustes.

[6]https://www.npmjs.com/package/bullmq

2.2. 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.

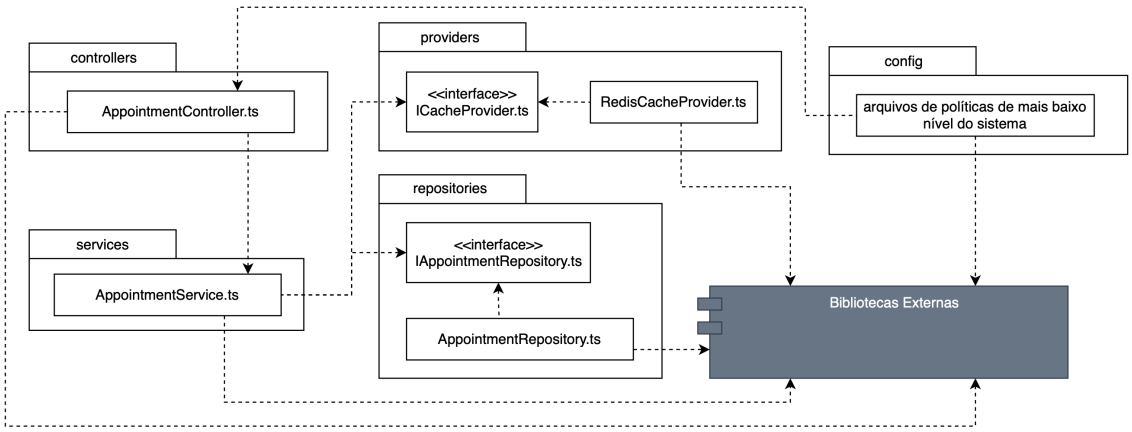

Figura 1. Diagrama arquitetural do sistema na arquitetura convencional.

Na Figura 1, é possível ter uma visão geral do projeto arquitetural do sistema em questão. É possível observar que os arquivos `AppointmentService.ts` e `AppointmentController.ts` tem alto acoplamento 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.

2.3. Implementação com Arquitetura Limpa

Após a modelagem dos elementos presentes na arquitetura convencional nas camadas da arquitetura limpa, foi realizada a identificação das decisões de projeto que não vão ao encontro do estabelecido pela arquitetura limpa. Importações de bibliotecas externas diretamente nas camadas de adaptadores de interface e casos de uso é uma dessas decisões de projeto.

Mais especificamente, as decisões de projeto erradas identificadas no projeto incluem: (i) uso direto da biblioteca *tsyringe* para injeção de dependências, que se mostrou inadequado para a arquitetura limpa devido ao uso de decoradores e contêineres espalhados em várias camadas; (ii) utilização direta das bibliotecas *date-fns* e *class-transform*, contrariando os princípios da arquitetura limpa que requerem camadas de abstração para facilitar futuras substituições; (iii) ausência de um arquivo da camada de entidades, que foi necessário criar e usar nos casos de uso para validar regras de domínio. Essas decisões exigiram significativas modificações para adequação à arquitetura limpa, resultando em um grande esforço de refatoração.

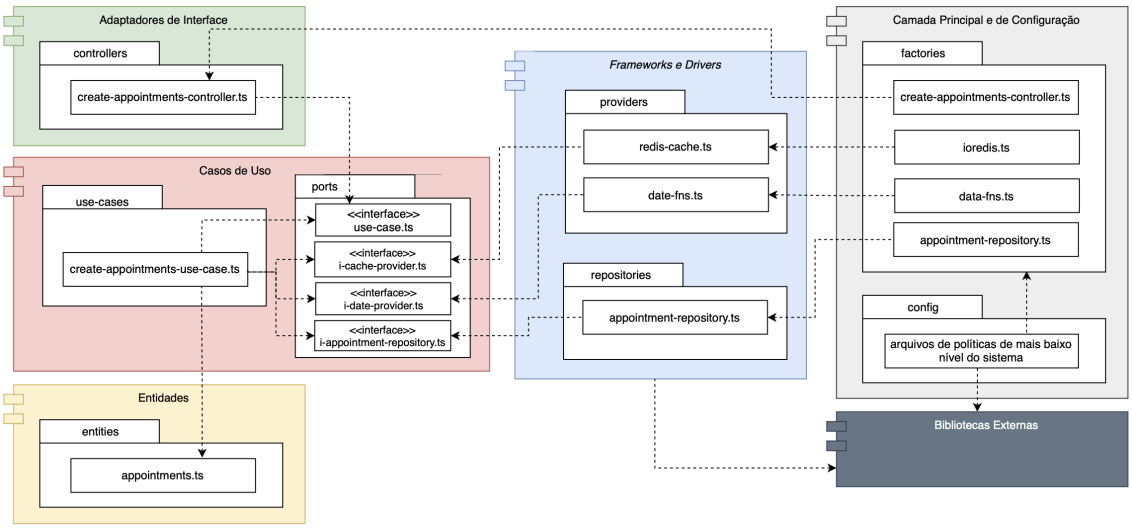

Figura 2. Diagrama arquitetural do sistema na arquitetural limpa.

Na Figura 2, 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.

3. 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 [17]. 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 [15]. 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 [15].

Esta seção conduz uma análise do esforço da troca de bibliotecas em um sistema convencional e em um com arquitetura limpa para tomadas de decisões por profissionais da área de desenvolvimento de software.

3.1. Metodologia

Realizou-se o seguinte conjunto de passos:

1. O primeiro autor deste artigo realizou sessões diárias de uma hora para a substituição de bibliotecas, a fim de evitar erros de medição devido à fadiga. A substituição começou pelo sistema com arquitetura limpa, seguida pelo sistema convencional, e as funcionalidades foram verificadas após cada substituição.
2. Foi utilizada a ferramenta *SonarQube*[7] para coletar dados de análise, focando nas métricas: linhas de código (LOC) [6], número de funções [3], número de classes [3] e número de arquivos [10].

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

3. A análise dos resultados enfatizou que a métrica LOC é a principal para avaliar o esforço, pois está diretamente relacionada à quantidade de linhas de código inseridas. No entanto, outras métricas também são relevantes e complementares, fornecendo uma visão sobre a modularidade e organização do código. Uma combinação dessas métricas e do tempo despendido é essencial para uma avaliação mais robusta.

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

Banco de dados relacionais ainda vigoram como o "padrão" para armazenamento de dados [14]. Como linguagens de programação atuais usualmente adotam o paradigma de orientação a objetos [4], 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 [2] .

Implementação Arquitetura Limpa: A conversão da biblioteca *typeorm* para *prisma* seguiu quatro fases:

1. Inclusão e configuração da biblioteca *prisma*.
2. Transformação do esquema do banco de dados para o esquema do *prisma* com o comando *prisma db pull*, facilitando a manipulação de dados com autocompletar em ambientes de desenvolvimento.
3. Implementação das operações de banco de dados com a biblioteca *prisma*, desenvolvendo uma classe que implementa a mesma interface utilizada nas implementações do *typeorm*.
4. Adequação das fábricas de repositórios, ajustando as importações das classes de repositório de *typeorm* para *prisma*, o que geralmente envolve a modificação de uma única linha de código.

Implementação Convencional: A conversão da biblioteca *typeorm* para *prisma* seguiu quatro fases:

1. Instalação e configuração da biblioteca *prisma*.
2. Transformação do esquema do banco de dados para o esquema do *prisma*.
3. Implementação das operações do banco de dados utilizando *prisma*, criando uma nova classe que implementa a interface utilizada nas implementações do *typeorm*.
4. Modificação do arquivo de fábrica para gerenciar a injeção de dependência dos repositórios, trocando importações de *typeorm* por *prisma*. A mudança incluiu ajustar as interfaces dos repositórios para usar os modelos do *prisma* e adaptar as classes de repositórios utilizadas nos testes de unidade e casos de uso, substituindo importações e tipagens do *typeorm* pelas do *prisma*.

Análise Quantitativa - Métricas: As métricas auferidas na Tabela 1 indicam – de forma sumarizada – 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 1. 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

Análise Quantitativa - Esforço: Por um lado, os tempos despendidos em cada uma das etapas reportados na Tabela 2 confirmam uma maior complexidade arquitetural na arquitetura limpa cujo desenvolvimento de fato se tornou 9,5% 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 significativamente menor, representando uma redução de 87,5% (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 2. 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**

3.3. 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) [8]. Diante disso, o Facebook lançou *GraphQL*[8] como uma nova abordagem de desenvolvimento de APIs que oferece eficiência e flexibilidade na integração de APIs [18].

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*.

[8]https://graphql.org

Implementação Arquitetura Limpa: A conversão da biblioteca *express* para *apollo-server* seguiu quatro fases[9]:

1. Configuração da biblioteca *apollo-server*.
2. Criação de diretivas de *middlewares* para autenticação e limitação de taxa de requisições usando bibliotecas adicionais: *@graphql-tools/utils*, *graphql-rate-limit-directive* e *graphql*.
3. Estruturação do *GraphQL* para cada caso de uso do sistema.
4. Substituição do servidor REST pelo *GraphQL*, ajustando apenas uma única linha nos códigos correspondentes.

Implementação Convencional: A conversão da biblioteca *express* para *apollo-server* seguiu seis fases:

1. Configuração da biblioteca *apollo-server*.
2. Criação do adaptador para servidor *GraphQL*, utilizando *args* e *context* para obter dados da requisição.
3. Criação de diretivas de *middlewares* para autenticação e limitação de requisições.
4. Criação das estruturas do *GraphQL*.
5. Troca do servidor REST pelo *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.

Análise Quantitativa - Métricas: Os dados apresentados na Tabela 3 mostram 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.

Análise Quantitativa - Esforço: Por outra perspectiva, a Tabela 4 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 3,6% 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 80% menor (5 minutos) em comparação com a implementação convencional (25 minutos). Portanto, os

[9]No início da substituição da biblioteca *express* pela *prisma*, foram identificadas incongruências no projeto, como o uso direto de *express* pelos controladores e a falta de generalidade do adaptador de requisições para atender REST e *GraphQL*. Correções foram feitas antes de prosseguir com a substituição.

Tabela 3. 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

dados coletados sugerem que a arquitetura limpa apresenta menor grau de acoplamento, o que resulta em uma troca de biblioteca mais eficiente.

Tabela 4. 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**

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

Fila de mensagens permite que servidores trabalhem de forma assíncrona de modo a continuarem operando sem interrupções [19]. 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 [5]. 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 [16].

Implementação Arquitetura Limpa: A conversão da biblioteca *bull* pela *bullmq* segue três etapas:

1. Inclusão da biblioteca *bullmq*.
2. Implementação das operações da fila com *bullmq*, desenvolvendo uma classe que implementa a interface usada nas implementações do *bull*.
3. Adequação das fábricas do *provider* da fila de mensagens, ajustando uma linha de código para substituir a importação da classe de repositório de *bull* para *bullmq*.

Implementação Convencional: A conversão da biblioteca *bull* pela *bullmq* segue três etapas:

1. Inclusão da biblioteca *bullmq*.

2. Implementação das operações de fila com *bullmq*.
3. Troca da biblioteca *bull* por *bullmq*, modificando duas linhas de código no arquivo de fábrica que gerencia a injeção de dependência dos *providers* usando *tsyringe*.

Análise Quantitativa - Métricas: De acordo com os dados apresentados na Tabela 5, 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.

Tabela 5. 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

Análise Quantitativa - Esforço: Por outro lado, as resultados apresentados na Tabela 6 indicam que, embora a implementação com arquitetura limpa tenha requerido um tempo de desenvolvimento 1,2% 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 66,7% menos tempo para realizar essa tarefa (5 minutos) em comparação com a convencional (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 6. 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**

3.5. Análise e Discussão

A análise dos dados revelou que a biblioteca *prisma* gerou menos artefatos de código fonte do que *typeorm* em ambos os sistemas. As bibliotecas *apollo-server* e *bullmq* apresentaram mais artefatos comparadas a *express* e *bull*, respectivamente, aumentando a complexidade do sistema.

A troca para *prisma* reduziu 39,2% mais linhas de código no sistema com arquitetura limpa (-461 LOC) em comparação com a convencional (-331 LOC). A troca para *apollo-server* também gerou menos esforço na arquitetura limpa, enquanto a convencional teve um aumento de 172,5% (139 LOC vs. 51 LOC). A troca para *bullmq* mostrou esforços equivalentes em ambos os sistemas. Pode-se afirmar 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*.

A implementação das bibliotecas no sistema com arquitetura limpa requereu mais tempo: *prisma* (115 min. vs. 105 min., 9,5% a mais), *apollo-server* (435 min. vs. 420 min., 3,6% a mais) e *bullmq* (425 min. vs. 420 min., 1,2% a mais). Contudo, a troca de bibliotecas no sistema com arquitetura limpa foi mais rápida: *prisma* (5 min. vs. 40 min., 87,5% mais rápido), *apollo-server* (5 min. vs. 25 min., 80% mais rápido) e *bullmq* (5 min. vs. 15 min., 66,7% mais rápido), indicando menor acoplamento e maior facilidade de manutenção.

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

4. Ameaças à validade

Dataset: Efetuada pelo primeiro autor deste artigo por meio da conversão de um sistema convencional. Embora algumas pessoas possam questionar tal transformação, (i) o primeiro autor desta pesquisa possui extensa especialização nas tecnologias empregadas, (ii) o projeto original do sistema foi elaborado por terceiros e (iii) a conversão está disponível em repositório público.

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 sempre primeiro realizada a substituição na arquitetura limpa. Ademais, algumas pessoas poderiam alegar questões de cansaço e fadiga, porém adotou-se sessões consecutivas de apenas uma hora por dia.

5. Conclusão

A arquitetura limpa busca tornar o código fácil de entender, manter e evoluir [12], mas aumenta a complexidade devido ao uso excessivo de interfaces e classes adicionais [13]. Este artigo avalia o esforço de troca de bibliotecas em sistemas com e sem arquitetura limpa, logo após criar um *dataset* com um mesmo sistema com duas implementações: arquitetura convencional e arquitetura limpa.

A substituição das bibliotecas *typeorm* por *prisma*, *express* por *apollo-server* e *bull* por *bullmq* mostrou que a troca para *prisma* gerou menos esforço na arquitetura limpa, reduzindo significativamente as linhas de código. A troca para *apollo-server* também gerou menos esforço na arquitetura limpa, porém com um aumento expressivo na quantidade de linhas de código na implementação convencional. A troca para *bullmq* mostrou esforços equivalentes em ambos os sistemas.

A implementação das bibliotecas no sistema com arquitetura limpa demandou mais tempo, contudo a troca foi mais rápida, indicando menor acoplamento e maior facilidade de manutenção. Futuras pesquisas podem explorar a troca de outras bibliotecas, incluir mais participantes na mensuração e utilizar outras métricas, além de comparar com sistemas de arquitetura mais simples.

Agradecimentos: Esta pesquisa é apoiada pela FAPEMIG (APQ-03513-18).

Referências

[1] Leonardo Henrique De Braz. Domain Driven Design: Vantagens e desvantagens em seu uso. B.sc. conclusion paper, Universidade Federal de Lavras, Trabalho de conclusão de curso de Ciência da Computação. Universidade Federal de Lavras, 2021.

[2] Felipe Buzzi. Prisma: uma das melhores coisas que já aconteceu no ecossistema? Disponível em: https://blog.rocketseat.com.br/prisma-uma-das-melhores-coisa-que-ja-aconteceu-no-ecossistema/, 2022. Acesso em: 13 fev. 2023.

[3] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[4] Samuel da Silva Feitosa and Rafaela Lunardi Comarella. Aprendendo conceitos de orientação a objetos usando as ferramentas Scratch e Snap! *Anais do Computer on the Beach*, 11(1):490–496, 2020.

[5] Godwin Ekuma. Asynchronous task processing in node.js with bull. Disponível em: https://blog.logrocket.com/asynchronous-task-processing-in-node-js-with-bull/, 2020. Acesso em: 13 fev. 2023.

[6] Norman E. Fenton. *Software Metrics: A Rigorous and Practical Approach*. CRC Press, 1999.

[7] Elton Fonseca. O que é orm? Disponível em: https://www.treinaweb.com.br/blog/o-que-e-orm, 2020. Acesso em: 13 fev. 2023.

[8] Guilherme Forte. Over-fetching and under-fetching: Rest apis' exhaustion signs. Disponível em: https://www.programmersinc.com/over-fetching-and-under-fetching-rest-apis-exhaustion-signs/, 2022. Acesso em: 13 fev. 2023.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] S. Henry and D. Kafura. Software metrics: a research evaluation. *IEEE Transactions on Software Engineering*, SE-7(6):576–590, 1981.

[11] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

[12] Robert C Martin, James Grenning, Simon Brown, Kevlin Henney, and Jason Gorman. *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall, 2018.

[13] Danilo Masotti. Clean architecture: descubra o que é e onde aplicar arquitetura limpa. Disponível em: https://www.zup.com.br/blog/clean-architecture-arquitetura-limpa, 2021. Acesso em: 13 jan. 2022.

[14] Cairo Noleto. Bancos de dados. Disponível em: https://blog.betrybe.com/tecnologia/bancos-de-dados/, 2021. Acesso em: 13 fev. 2023.

[15] Jacqueline Oliveira. Log4j: entenda mais sobre a vulnerabilidade do bug. Disponível em: https://www.alura.com.br/artigos/log4j-entenda-sobre-vulnerabilidade, 2021. Acesso em: 13 fev. 2023.

[16] Marudhu Pandiyan. Bullmq — message queue in node.js. Disponível em: https://medium.com/@marudhupandiyang/bullmq-message-queue-in-node-js-bca24bac7f8a, 2023. Acesso em: 13 fev. 2023.

[17] Priyadarshi Tripathy and Kshirasagar Naik. *Software evolution and maintenance: a practitioner's approach*. John Wiley & Sons, 2014.

[18] Bruna Vidanya. Graphql vs. rest: Qual o melhor para o desenvolvimento de api? Disponível em: https://www.hostinger.com.br/blog/graphql-vs-rest-qual-o-melhor-para-o-desenvolvimento-de-api/, 2021. Acesso em: 13 fev. 2023.

[19] Larry | Peng Yang. System design - message queues. Disponível em: https://medium.com/must-know-computer-science/system-design-message-queues-245612428a22, 2020. Acesso em: 13 fev. 2023.