Lucas de Rossi Bernardi

Estudo Comparativo da Utilização de Design Patterns no Desenvolvimento de Uma API REST com TypeScript

Passo Fundo

2022

Lucas de Rossi Bernardi

Estudo Comparativo da Utilização de Design Patterns no Desenvolvimento de Uma API REST com TypeScript

Monografia apresentada ao Curso de Tecnologia em Sistemas para Internet do Instituto Federal Sul-rio-grandense, Câmpus Passo Fundo, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SUL-RIO-GRANDENSE - CÂMPUS PASSO FUNDO

Orientador: Prof. Me. André Fernando Rollwagen

Passo Fundo
2022

Resumo

A utilização de padrões de projeto como roteiro para a solução de problemas no âmbito do desenvolvimento de software é uma prática comum entre desenvolvedores. Porém, estudos apontam que muitos destes padrões acabam aumentando a complexidade geral do projeto, por meio da adição de novas classes, métodos e relacionamentos ao código. Desta forma, o presente trabalho teve como objetivo analisar os impactos causados pela aplicação dos padrões de projeto no quesito da qualidade do código de um sistema. Para isso, foram analisadas duas diferentes versões do mesmo projeto, onde a primeira foi desenvolvida sem a utilização dos padrões e posteriormente refatorado em uma segunda versão contendo padrões estruturais, criacionais e comportamentais. A análise foi feita utilizando a ferramenta SonarQube, que identifica métricas relacionadas ao tamanho, complexidade, manutenibilidade, segurança, dentre outros atributos relacionados a base de código do sistema. Como resultados, notou-se que os padrões de projeto utilizados impactaram de forma positiva na qualidade do código, resultando em um sistema de fácil manutenção e auxiliando na aplicação dos 5 princípios de código limpo da Programação Orientada a Objetos ou SOLID.

Palavras-chave: Padrões de Projeto, Princípios SOLID, API REST, Qualidade de Código.

Abstract

The use of design patterns as a roadmap for solving problems in software development is a common practice among developers. However, studies show that many of these patterns end up increasing the overall complexity of the project, through the addition of new classes, methods and relationships to the code. The present work aimed to analyze the impacts caused by the application of design patterns in terms of code quality. Two different versions of the same project were analyzed, where the first was developed without the use of patterns and later refactored into a second version containing structural, creational and behavioral patterns. The analysis was performed using SonarQube, which identifies metrics related to size, complexity, maintainability, security, among other attributes related to the system's code base. As a result, it was noted that the design patterns used had a positive impact on the quality of the code, resulting in an easy-to-maintain system and helping to apply the 5 clean code principles of Object Oriented Programming or SOLID.

Keywords: Design Patterns, SOLID Principles, REST API, Code Quality.

Lista de ilustrações

Lista de tabelas

Lista de códigos fonte

Lista de abreviaturas e siglas

API	Application Programming Interface
CI/CD	Continuous Integration/Continuous Delivery
CRUD	Acrônimo do inglês Create, Read, Update and Delete
DIP	Dependency Inversion Principle
DTO	Data Transfer Object
HTTP	Hypertext Transfer Protocol
ISP	Interface Segregation Principle
JS	JavaScript
JWT	JSON Web Token
LSP	Liskov Substitution Principle
MVC	Acrônimo de Model-View-Controller
OCP	Open-Closed Principle
OMT	Object-Modeling Technique
POO	Programação Orientada a Objetos
REST	Representational State Transfer
SOLID	Acrônimo de Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle e Dependency Inversion Principle
SQL	Structured Query Language
SRP	Single Responsibility Principle
TS	TypeScript
UML	Unified Modeling Language

Sumário

1 Introdução

O desenvolvedor é responsável por criar um plano de construção de software que esteja dentro das condições estabelecidas pelo cliente (FERNANDES, 2003). Ou seja, o papel do desenvolvedor é encontrar uma solução que atenda às especificidades requeridas de cada software solucionando e resolvendo problemas da melhor forma possível.

Shvets (2021) afirma que os padrões de projeto são sugestões de solução para problemas rotineiros no âmbito do desenvolvimento de software. Esses padrões foram especificados e exemplificados formalmente em Gamma et al. (1995) para que possam ser utilizados em diferentes contextos e aplicados no desenvolvimento dos mais variados projetos, garantindo assim uma arquitetura limpa com foco em seguir os principais Princípios da Programação Orientada a Objetos (POO).

O conceito de padrões de projeto foi observado e introduzido na arquitetura por Alexander (1977) no livro: "A Pattern Language: Towns, Buildings, Construction" e posteriormente o termo foi incorporado ao ambiente de desenvolvimento de software na criação de *design patterns*.

Os 5 princípios da POO foram criados por Martin, Newkirk e Koss (2003) e tem como objetivo estabelecer bases que auxiliam o desenvolvedor a escrever códigos limpos, tornando todo o ecossistema de desenvolvimento do software em questão um ambiente seguro de se trabalhar, onde cada componente tem uma única responsabilidade e o acoplamento de código é reduzido, facilitando a refatoração e reaproveitamento de funcionalidades (PAIXAO, 2019).

Tendo em vista que alguns padrões de projeto acabam aumentando a complexidade geral do código introduzindo novas classes e métodos com diferentes propósitos (GAMMA et al., 1995), o presente trabalho buscou responder a seguinte questão: Quais são os efeitos causados na qualidade geral do código de um sistema ao se aplicar determinados padrões de projeto?

Para isto, foi desenvolvida uma *Application Programming Interface* (API) *Representational State Transfer* (REST) de controle de estoque utilizando o ExpressJS[1] em conjunto ao TypeScript[2] no ambiente de execução NodeJs[3] persistindo dados através do PostgreSQL[4] com o auxílio do Knex[5] para realizar a conexão entre o banco de dados e a API.

[1] https://expressjs.com/
[2] https://www.typescriptlang.org/
[3] https://nodejs.org/
[4] https://www.postgresql.org/
[5] https://knexjs.org/

A primeira versão do sistema foi implementada sem a utilização dos padrões de projeto e posteriormente refatorada com foco na aplicação dos mesmos. Ambas as versões foram comparadas utilizando a ferramenta SonarQube[6] que gerou métricas com relação aos atributos da base de código de cada uma.

1.1 Objetivos

1.1.1 Objetivo Geral

O presente trabalho teve como objetivo realizar um estudo dos impactos gerados na qualidade do código após a aplicação de *design patterns* em um sistema que inicialmente foi desenvolvido sem a utilização dos mesmos.

1.1.2 Objetivos Específicos

- Apresentar as consequências da implementação de *design patterns*;
- Realizar elicitação de requisitos para o desenvolvimento da API REST de controle de estoque;
- Analisar a qualidade do código de ambas versões do sistema;
- Comparar os resultados obtidos e impactos gerados pela refatoração na qualidade geral da aplicação;
- Elencar os índices e vantagens da utilização dos padrões de projeto no desenvolvimento do sistema.

1.2 Organização do Trabalho

A organização do trabalho foi feita em 7 capítulos, sendo eles:

- Capítulo 2 - Apresenta a base teórica contendo os principais frameworks e bibliotecas utilizadas no desenvolvimento do sistema, bem como uma introdução aos princípios SOLID e *design patterns* utilizados na refatoração da aplicação;
- Capítulo 3 - Discute os trabalhos relacionados a esta monografia, com foco na comparação e análise das consequências da implementação de padrões de projeto e/ou princípios SOLID;
- Capítulo 4 - Apresenta a metodologia utilizada para desenvolver o trabalho;

[6] https://www.sonarqube.org/

- Capítulo 5 - Detalha o processo de desenvolvimento do estudo de caso;
- Capítulo 6 - Apresenta de qual forma foram obtidas as métricas e as consequências resultantes da aplicação dos padrões de projeto e princípios SOLID;
- Capítulo 7 - Conclui o trabalho e detalha possíveis melhorias a serem aplicadas no futuro.

2 Referencial Teórico

Neste capítulo serão apresentadas as principais bibliotecas e frameworks utilizados no desenvolvimento deste trabalho, bem como os conceitos relacionados aos princípios SOLID e *design patterns* aplicados.

2.1 Bibliotecas e Frameworks

2.1.1 TypeScript

TypeScript (TS) é um superconjunto sintático estrito da linguagem JavaScript (JS) desenvolvido pela Microsoft. Seu principal foco é adicionar a tipagem estática ao JS, além de outras funcionalidades comumente encontradas em linguagens orientadas a objetos, como interfaces, classes abstratas, dentre outros (TYPESCRIPT, 2022).

A utilização do TS garante um processo consistente de desenvolvimento se comparado ao mesmo utilizando apenas JS, isso se dá pelo fato de que toda a informação trafegada na aplicação já tem um formato esperado, o que previne os erros mais comuns encontrados nos sistemas JavaScript. Já que todas as informações são tipadas, também é possível ter um *intellisense*[1] preciso se combinado a um bom editor de texto.

Conforme explicado por Bierman, Abadi e Torgersen (2014) a curva de aprendizado da tecnologia tende a ser grande, ainda mais para pessoas que não conhecem uma linguagem tipada por padrão, no entanto, quando dominada as funcionalidades básicas da ferramenta, a curva tende a diminuir.

2.1.2 Knex

O Knex é um *Query Builder*, ou seja, ele permite a criação dinâmica de *queries* no formato *Structured Query Language* (SQL)[2]. Seu diferencial está na adaptabilidade, pois a biblioteca consegue utilizar diferentes clientes para realizar a conexão com a base de dados sem a necessidade de alterar as *queries* já desenvolvidas (KNEX, 2022). Logo, caso seja necessário substituir o banco de dados da aplicação, as especificidades de cada um não serão um empecilho, visto que o Knex já realiza as devidas alterações nas *queries* de acordo com o cliente configurado.

[1] Intellisense: Recurso de autocompletar de código com base no contexto em alguns ambientes de programação que acelera o processo de codificação de aplicativos, reduzindo erros de digitação e outros erros comuns.

[2] SQL: Linguagem de pesquisa declarativa padrão para banco de dados relacional.

2.1.3 Express

Segundo Mardan (2014), o express é um framework que foi criado a partir do módulo *Hypertext Transfer Protocol* (HTTP) nativo do NodeJs e fornece recursos mínimos para construção de serviços web escaláveis através de um conjunto de ferramentas robustas e *middlewares* que podem ser adaptados de acordo com a necessidade.

O framework tem como base a ideologia minimalista e flexível (EXPRESS, 2022), atribuindo a responsabilidade de planejar e organizar a estrutura da aplicação ao desenvolvedor. Sendo assim, o express pode ser utilizado tanto com modelos arquitetônicos clássicos como o MVC quanto modelos customizados que atendam um propósito específico.

2.2 SOLID

O SOLID é uma sigla mnemônica em inglês para cinco princípios de projeto que tem como objetivo guiar o desenvolvimento de software por meio de um conjunto de regras e bons padrões a serem adotados com o foco em tornar o projeto compreensivo, flexível, e sustentável (SHVETS, 2021).

Por mais que sejam voltados à POO, os princípios também podem ser adaptados para outros paradigmas computacionais como a programação funcional. A presente seção visa introduzir os cinco princípios e quais as vantagens de sua aplicação.

2.2.1 *Single Responsibility Principle*

O *Single Responsibility Principle* (SRP), propõe que uma classe deve ser especializada em um único assunto e possuir apenas uma responsabilidade, como descrito por Martin, Newkirk e Koss (2003), “Reúna as coisas que mudam pelas mesmas razões. Separe as coisas que mudam por diferentes razões”.

A violação do SRP pode resultar em falta de coesão, alto acoplamento entre classes, dificuldades na implementação de testes automatizados e dificuldade para reaproveitar o código (MARTIN et al., 2018).

2.2.2 *Open-Closed Principle*

Conforme explicado por Martin, Newkirk e Koss (2003), o *Open-Closed Principle* (OCP) é caracterizado pela seguinte frase: “Um módulo deve estar aberto para extensão, mas fechado para modificação”. Ou seja, ao invés de modificar uma classe que já foi devidamente testada e está sendo utilizada por outros componentes no sistema, o correto é estender a mesma e implementar a nova funcionalidade na extensão, evitando que possíveis *bugs* da nova implementação atrapalhem os outros componentes relacionados a classe que anteriormente estava funcionando conforme o esperado.

O OCP é um dos principais conceitos aplicados na arquitetura de software limpa, seu objetivo segundo Martin et al. (2018) é tornar o sistema fácil de estender sem prejudicar o que já foi desenvolvido. Se aplicado corretamente, esse princípio pode trazer flexibilidade, reutilização e manutenibilidade ao código.

2.2.3 *Liskov Substitution Principle*

Martin, Newkirk e Koss (2003) definem o *Liskov Substitution Principle* (LSP) com "Os subtipos devem ser substituíveis por seus tipos básicos". Ou seja, se B é um subtipo de A, então, B pode ser trocado completamente por A sem que seja necessário fazer alterações.

A implementação correta do LSP, possibilita a utilização de polimorfismo e *Duck typing*[3] de forma coesa, garantindo que resultados inesperados não ocorram.

2.2.4 *Interface Segregation Principle*

O *Interface Segregation Principle* (ISP) define que uma classe não deve ser forçada a implementar interfaces e métodos que não irão utilizar, ou seja, as interfaces devem ser criadas com um propósito específico ao invés de genérico e único (MARTIN; NEWKIRK; KOSS, 2003).

2.2.5 *Dependency Inversion Principle*

Martin et al. (2018) explica que o *Dependency Inversion Principle* (DIP) consiste em uma regra que diz que classes concretas devem depender apenas de interfaces abstratas, ou contratos que definem quais métodos deverão ser implementados pelas dependências, sendo assim, a classe não fica acoplada a apenas uma implementação.

Esse princípio está diretamente ligado ao OCP, pois ao organizar as classes para que estejam abertas para extensão, elas obrigatoriamente devem depender de uma interface ao invés de uma classe concreta.

Ao implementar corretamente o DIP, o processo de escrita de testes é facilitada, visto que não serão necessários grandes mocks[4] para que a unidade seja testada de forma eficaz.

[3] Duck typing: Estilo de codificação de linguagens dinamicamente tipadas onde o tipo de uma variável não importa, contanto que seu comportamento seja o desejado. O nome "tipagem de pato" vem da expressão "se anda como pato, nada como um pato e faz quack como um pato, então provavelmente é um pato" (AYLAN, 2019)

[4] Mocks: Objeto falso que simula o comportamento de uma classe específica ou real para que o teste seja focado apenas na unidade que importa.

2.3 *Design Patterns*

Um padrão de projeto pode ser definido da seguinte forma:

> Cada padrão descreve um problema que ocorre repetidamente em nosso ambiente e, em seguida, descreve o núcleo da solução para esse problema, de forma que você possa usar essa solução um milhão de vezes, sem nunca fazer da mesma maneira duas vezes. (ALEXANDER, 1977 apud GAMMA et al., 1995).

Os padrões de projeto podem ser classificados como: Criacionais, que fornecem mecanismos de criação de objetos que aumentam a flexibilidade e a reutilização de código; Estruturais, que definem como montar objetos e classes em estruturas maiores, mantendo estruturas flexíveis e eficientes; Comportamentais, que gerenciam a comunicação de forma eficiente entre objetos (SHVETS, 2021).

Gamma et al. (1995) também afirmam que para descrever um padrões de projeto é necessário definir principalmente os seguinte itens:

- Nome: Deve conter a essência do mesmo descrita de forma sucinta;
- Classificação: Definir em qual classificação o padrão se enquadra;
- Intenção: Uma frase curta que responda às perguntas: O que o padrão de projeto faz? Qual é a sua razão e intenção? Qual problema de design ele aborda?;
- Motivação: Um cenário que ilustra um problema de design e como as estruturas de classe e objeto no padrão resolvem o problema. O cenário irá ajudá-lo a entender a descrição mais abstrata do padrão;
- Aplicabilidade: Apresentar quais são as situações em que o padrão de projeto pode ser aplicado, exemplos de problemas de design que o padrão pode solucionar e como reconhecer essas situações;
- Estrutura: Uma representação gráfica das classes no padrão utilizando uma notação baseada na *Object-Modeling Technique* (OMT)[5];
- Exemplo de Código: Fragmentos de pseudocódigo ilustrando como seria feita a implementação do padrão.

Em Gamma et al. (1995), são apresentados vinte e três padrões de projeto, porém, apenas três foram utilizados no estudo de caso deste trabalho, dentre eles estão o *Adapter*, *Factory Method* e *Template Method* que foram selecionados observando os problemas que cada um propõem solucionar com base nos problemas encontrados no desenvolvimento da

[5] Atualmente a forma mais utilizada para definir a estrutura dos padrões de projeto é a *Unified Modeling Language* (UML) (SHVETS, 2021).

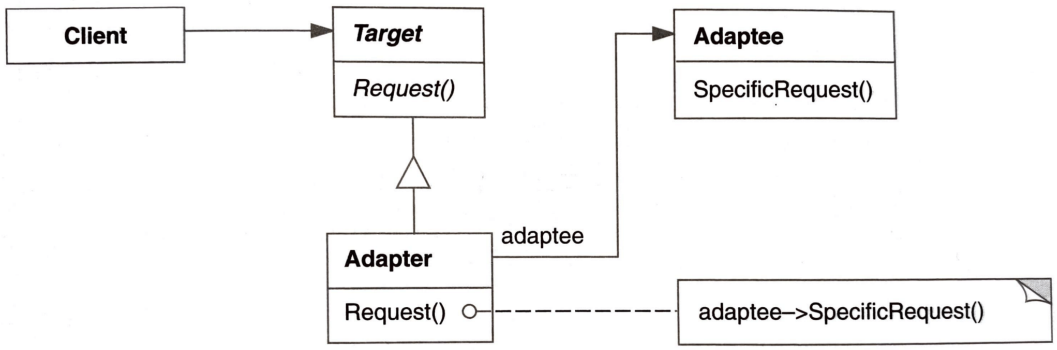
primeira versão da API. Essa seção irá apresentar a intenção, estrutura, aplicabilidade e consequências da implementação dos *design patterns* supracitados.

2.3.1 *Adapter*

O *Adapter* é um padrão de projeto estrutural que tem como intenção "Converter a interface de uma classe em outra interface, esperada pelos clientes. O *Adapter* permite que classes com interfaces incompatíveis trabalhem em conjunto o que, de outra forma, seria impossível" (GAMMA et al., 1995).

Ao observada a Figura 1, é possível perceber que esse padrão de projeto possui os seguintes participantes: *Target*, que define a interface específica do domínio que *Client* usa; *Client*, responsável por utilizar objetos compatíveis com a interface *Target*; *Adaptee*, que define uma interface existente que necessita ser adaptada; *Adapter*, responsável por adaptar a interface de *Adaptee* para *Target* (GAMMA et al., 1995).

Figura 1 – Estrutura do Padrão de Projeto Adapter



Fonte: (GAMMA et al., 1995)

Ao adotar o padrão *Adapter*, Shvets (2021) afirma que a complexidade do código tende a aumentar, pois novas classes e interfaces serão adicionadas ao projeto para solucionar o problema, quando em certos casos, o mais simples seria mudar o *Adaptee* para atender as necessidades do *Client*, sem a utilização de um *Adapter*.

Por outro lado, a utilização desse padrão garante a aplicação do SRP, pelo fato da lógica de adaptação dos objetos estar centralizada no *Adapter*. Outro princípio aplicado com esse padrão é o OCP, pois *Client*, *Target* e *Adaptee* seguirão com suas implementações intactas enquanto o *Adapter* lida com as incompatibilidades (SHVETS, 2021).

2.3.2 *Factory Method*

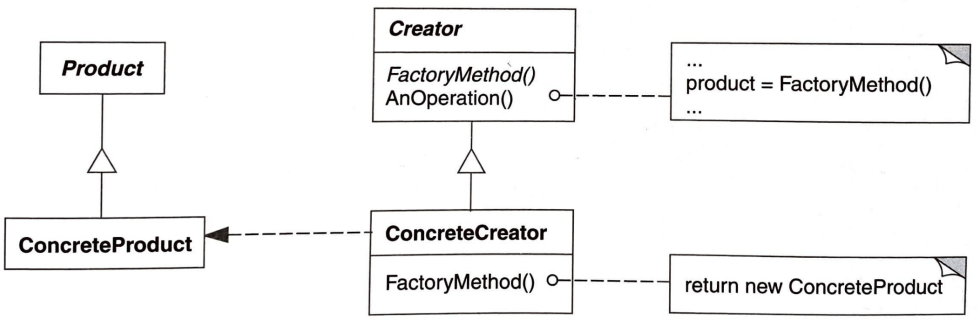
O *Factory Method* é um padrão de projeto criacional, ou seja, lida com a criação de objetos e classes. Seu objetivo principal é ocultar a lógica de instanciação das classes

no código cliente, passando essa responsabilidade a classe fábrica que internaliza a tarefa lidando com as dependências e especificidades de cada classe (SHVETS, 2021).

Sendo assim, Gamma et al. (1995) definem a intenção deste padrão como "Definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar. O *Factory Method* permite adiar a instanciação para as subclasses".

Gamma et al. (1995) também afirmam que a implementação do *Factory Method* auxilia na aplicação do OCP, pois novas funcionalidades estenderão da classe de construção e não da implementação concreta, mantendo a base da funcionalidade inalterada. O SRP também será aplicado, visto que a responsabilidade de fabricar novas instâncias ficará centralizada na classe fábrica, resultando no desacoplamento do código.

Figura 2 – Estrutura do Padrão de Projeto *Factory Method*



Fonte: (GAMMA et al., 1995)

Contudo, este padrão de projeto pode ser utilizado para desacoplar o código de criação do código de utilização de classes, permitir que subclasses decidam a lógica de fabricação dos objetos de acordo com a necessidade e por fim, eliminar a duplicação de código no processo de criação de objetos.

2.3.3 *Template Method*

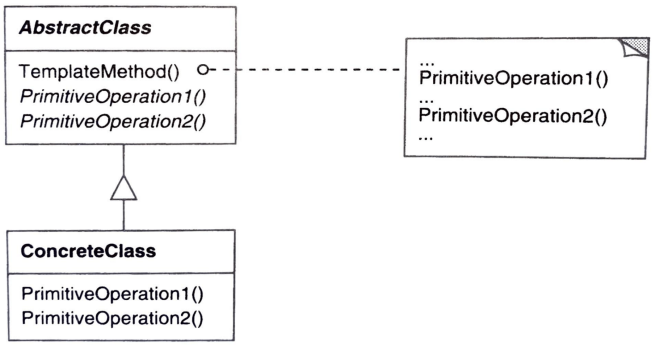
Este é um padrão comportamental e segundo Gamma et al. (1995), sua intenção é: "Definir o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses. O *Template Method* permite que as subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura".

A aplicação deste padrão é recomendada em casos onde a herança está sendo aplicada para alterar apenas pequenos trechos de código de um algoritmo, mas a estrutura geral do mesmo é mantida (SHVETS, 2021).

Na Figura 3 podemos perceber que a execução das operações está contida na classe abstrata e apenas as operações podem ou não ser subscritas pela classe concreta, sem que

ocorra alteração no processo de execução das mesmas.

Figura 3 – Estrutura do Padrão de Projeto Template Mehod



Fonte: (GAMMA et al., 1995)

Como consequências da implementação deste padrão, Gamma et al. (1995) cita a fácil manutenção do algoritmo e a remoção da duplicação de código, visto que a lógica estará centralizado na classe *Template*.

2.4 SonarQube

O SonarQube é uma plataforma que realiza a inspeção estática de código fonte, com foco em detectar *bugs*, *code smells*[6], falhas de segurança, cobertura de testes, duplicação e complexidade de código. Atualmente a plataforma suporta a análise de vinte e nove diferentes linguagens de programação, além de possuir integração com ferramentas de CI/CD e versionamento de código (SONARSOURCE, 2022).

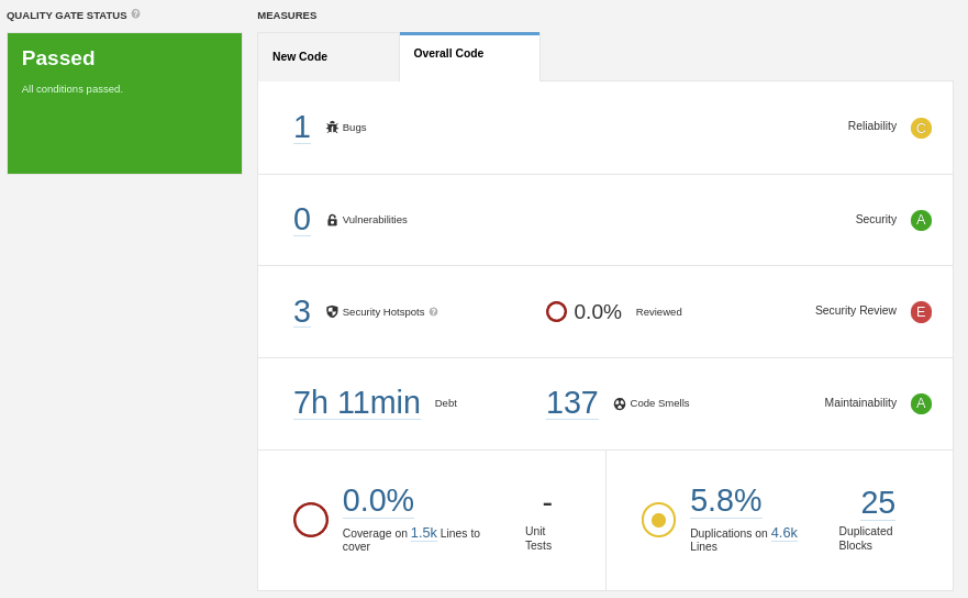
Após realizar a inspeção, a plataforma gera relatórios sobre os diferentes aspectos da aplicação atribuindo notas de A até E. Conforme apresentado por SonarSource (2022) os aspectos analisados são:

- Confiabilidade: Diretamente relacionada ao número e nível de *bugs*, ou seja, caso identificado um *bug* pequeno a nota será B, porém, se o *bug* for crítico a nota será D. Desta forma, quanto menor for a nota, maior será a chance de ocorrer comportamentos inesperados no sistema;
- Segurança: Utiliza os mesmos princípios da Confiabilidade, porém aplicada a vulnerabilidades do código;

[6] Code Smell: Trechos ou características de código que, possivelmente podem causar problemas futuros (FOWLER, 2018).

- Manutenibilidade: Essa métrica utiliza como base o número de *code smells* encontrados no código em conjunto ao tempo estimado para resolvê-los. Sendo assim, quanto menor a nota, mais difícil será manter a base de código sólida;
- Cobertura: Busca analisar quantas linhas e cenários de código foram validadas com os testes unitários;
- Duplicações: Apresenta o número de arquivos, linhas e blocos duplicados. Atributos que segundo Singh e Gautam (2016) impactam na confiabilidade e manutenibilidade do código;
- Tamanho: Apresenta o número de linhas, instruções, funções, classes, arquivos e comentários presentes no projeto;
- Complexidade: Essa métrica utiliza fórmulas matemáticas em conjunto ao número de caminhos possíveis que o código pode tomar para identificar o quão difícil é entende-lo e testá-lo completamente.

Figura 4 – Dashboard SonarQube



Fonte: Do autor, 2022

Analisando a Figura 4, percebe-se que o SonarQube apresenta inicialmente uma visão geral do projeto, contendo informações relacionadas a diferentes aspectos do código. A primeira informação é a quantidade de *bugs* encontrados, bem como a nota referente a confiabilidade. Em seguida é apresentada a quantidade de vulnerabilidades e a nota de segurança do sistema. Também está presente na *Dashboard* a quantidade de *code smells* e o tempo calculado para resolvê-los. Por fim, é apresentado métricas relacionadas a cobertura de testes e duplicação de código.

3 Trabalhos Relacionados

O presente capítulo tem como objetivo apresentar e discutir outros trabalhos científicos que tenham objetivos similares a este ou que de alguma forma contribuem com estudos relevantes para o desenvolvimento desta monografia.

3.1 Trodin (2021)

O trabalho visa realizar um estudo comparativo das consequências de se aplicar os princípios SOLID em um projeto utilizando ReactJs. A comparação foi feita com uma aplicação em duas versões diferentes, sendo a primeira delas um exemplo de calculadora disponibilizado pelo ReactJs e a segunda uma refatoração feita pela autora no mesmo sistema com foco em aplicar os princípios SOLID utilizando POO (TRODIN, 2021).

Para estabelecer as métricas de sua refatoração, Trodin (2021) utiliza os estudos de McCabe (1976) e Chidamber e Kemerer (1994), onde são definidas fórmulas e conceitos para medir a qualidade do código com base na sua complexidade e métricas de baixo nível, como acoplamento entre classes de objetos, número de filhos, profundidade da árvore de herança, dentre outros indicativos relevantes da POO (JORGENSEN, 2013 apud TRODIN, 2021).

A principal diferença entre os trabalhos é que Trodin desenvolve um estudo de caso voltado ao Front-End utilizando JS com PropTypes[1] refatorando-o com foco em aplicar os princípios SOLID, enquanto o presente estudo visa desenvolver uma API REST com TypeScript aplicando padrões de projeto e SOLID.

O trabalho apresenta os resultados obtidos com a aplicação dos princípios SOLID demonstrando exemplos e descrevendo as principais dificuldades e facilidades referente ao processo de refatoração do sistema fazendo uso de conceitos e descrições de Martin, Newkirk e Koss (2003).

Por fim, o estudo conclui que o código no aplicativo reescrito é melhor do que no aplicativo original, pelo fato de ser mais escalável e fácil de manter, além de ter obtido resultados melhores nas métricas utilizadas (TRODIN, 2021). Sua principal contribuição com o presente trabalho foi a metodologia utilizada, bem como a forma como foi feita a analise de quais princípios SOLID foram aplicados com a refatoração.

[1] PropTypes: Mecanismo que realiza a checagem estática de tipos dos atributos passados para componentes (REACT, 2022)

3.2 Singh e Gautam (2016)

Nesse estudo são apresentados os impactos gerados por cada etapa do desenvolvimento de software em aspectos da qualidade de código. As etapas analisadas foram de Análise de Requisitos, Design de Software, Desenvolvimento e Testes.

Conforme descrito por Singh e Gautam (2016), a etapa que mais causa impacto na qualidade do software é a de design, que é dividida em quatro módulos, sendo eles: Arquitetura de Software; Modularidade; Padrões de Projeto; Design de Componentes do Software.

Tabela 1 – Relação entre os Atributos de Design de Software e Atributos de Qualidade de Software

Módulo de Design	**Atributos de Qualidade**
Arquitetura de Software	adaptabilidade, funcionalidade, segurança, eficiência, portabilidade, interoperabilidade, reusabilidade, manutenibilidade, confiabilidade, usabilidade, desempenho e evolução
Modularidade	mutabilidade e adaptabilidade
Padrões de Projeto	reusabilidade, composição, completude, manutenibilidade, estabilidade, compreensão, flexibilidade, simplicidade, e expansibilidade
Design de Componentes do Software	responsabilidade, estabilidade e modularidade

Fonte: (SINGH; GAUTAM, 2016)

O trabalho é concluído apresentando a importância da manutenibilidade de código, que é o atributo mais relevante da qualidade de software, pois consiste no processo de refinamento do sistema para que o mesmo continue atendendo as necessidades do negócio, além de impactar no correção de falhas e melhoria de desempenho (SINGH; GAUTAM, 2016).

Os dados obtidos no quesito de atributos impactados pela aplicação de padrões de projeto auxiliou na análise dos resultados fornecidos pela aplicação da ferramenta SonarQube em abas as versões da API, visto que também são geradas métricas relacionadas a outros atributos de qualidade de código que não são impactados pela aplicação de *design patterns*.

3.3 Khomh e Guéhéneuc (2008)

Esse trabalho visa analisar os impactos do uso de *design patterns* nos quesitos de manutenibilidade e evolução de sistemas. Khomh e Guéhéneuc (2008) afirmam que o

processo de desenvolvimento e manutenção de software são atividades manuais realizadas por desenvolvedores, portanto, o método utilizado para realizar o estudo foi um questionário destinado aos mesmos, para que avaliem de forma quantitativa e qualitativa os impactos gerados na qualidade do sistema.

Os impactos causados pelos padrões de projeto na qualidade de software está diretamente relacionado a necessidade do mesmo ser aplicado e como foi feita a implementação (KHOMH; GUÉHÉNEUC, 2008). Sendo assim, é possível que a aplicação errônea de um *design pattern* resulte na perda de alguns atributos de qualidade.

No questionário foram abordados tópicos sobre o design, implementação e execução do software, obtendo notas de A a F, sendo A muito positivo e F não aplicável. Os resultados apresentados foram coletados a partir da aplicação do questionário com 20 desenvolvedores com sólida experiência na área.

Para apresentar os resultados, os autores organizaram uma tabela contendo todos os padrões de projeto disponibilizados por Gamma et al. (1995), relacionando-os aos atributos do questionário, onde cada característica possui um indicador demonstrando se o padrão impacta positivamente (+) ou negativamente (-) ao determinado aspecto.

Tabela 2 – Estimativa do impacto dos padrões de projeto nos três atributos de qualidade

Padrão de Projeto	**Expansibilidade**	**Compreensibilidade**	**Reutilização**
A. Factory	+	-	+
Builder	+	+	-
F. Method	+	-	+
Prototype	+	+	+
Singleton	-	+	-
Adapter	+	-	+
Bridge	+	+	-
Composite	+	+	+
Decorator	+	-	-
Facade	+	+	-
Flyweight	-	-	-
Proxy	-	-	+
Ch. Of. Resp	+	-	+
Command	+	-	-
Interpreter	+	+	+
Iterator	+	+	+
Mediator	+	+	-
Memento	-	-	-
Observer	+	-	+
State	+	+	-
Strategy	+	+	-
T. Method	+	-	+
Visitor	+	-	-
	19 + / 4 -	11 + / 12 -	11 + / 12 -

Fonte: (KHOMH; GUÉHÉNEUC, 2008)

Analisando a Tabela 2, Khomh e Guéhéneuc (2008) concluem que os padrões de projeto nem sempre melhoram a qualidade dos sistemas. Sendo necessário utilizá-los com cautela durante o desenvolvimento, pois podem impactar negativamente na manutenção e evolução do sistema, sendo essa a principal contribuição com o presente trabalho.

4 Metodologia

A fim de atingir os objetivos supracitados, foram definidas etapas de pesquisa e desenvolvimento sobre os temas relacionados ao presente trabalho. O primeiro momento foi centrado no estudo dos princípios SOLID que segundo Shvets (2021) são a base para fazer dos projetos de software algo mais compreensivo, flexível, e sustentável. Também foi necessário estudar sobre os padrões de projeto catalogados em Gamma et al. (1995), compreendendo suas motivações, problemas resolvidos, formas de aplicação e ocasiões onde os mesmos podem ser utilizados.

O segundo momento foi utilizado para encontrar formas de se medir os impactos causados por uma refatoração em um sistema, buscando quais soluções foram utilizadas para realizar o comparativo em trabalhos relacionados e se existiam ferramentas que automatizam o processo e apresentem métricas relevantes quanto a qualidade do código nas duas diferentes versões. Neste momento também foi feito o levantamento de requisitos do estudo de caso para ser utilizado como base no desenvolvimento.

Em seguida, no terceiro momento foi feito o desenvolvimento da primeira versão da API REST sem utilizar os padrões de projeto, seguindo todos os documentos gerados com o levantamento de requisitos e utilizando conceitos arquiteturais apresentados por Martin et al. (2018) a fim de tornar a arquitetura da aplicação limpa e escalável, além de facilitar o desacoplamento de código e aplicação dos princípios SOLID. Para garantir que o sistema atenda os requisitos especificados, foram definidos casos de teste com operações básicas para realizar os fluxos mais críticos da API. Os testes foram executados utilizando a ferramenta Postman[1].

No quarto momento, o foco foi a refatoração da primeira versão da API, buscando identificar possíveis melhorias a serem aplicadas no código e situações onde os padrões de projeto estudados no primeiro momento poderiam ser utilizados de forma coerente. Após o processo de identificação, foi feito o desenvolvimento das melhorias com base na estrutura e exemplos disponibilizados na descrição dos *design patterns*.

Posteriormente, no quinto momento foi aplicada a ferramenta SonarQube para obter métricas relacionadas a qualidade do código em ambas as versões e feito um comparativo dos ganhos obtidos com a refatoração.

Por fim, os resultados obtidos foram elencados a presente monografia, com foco em divulgar os impactos gerados com a aplicação dos padrões de projeto supracitados e divulga-los por meio de publicações em periódicos e congressos.

[1] https://www.postman.com/

5 Desenvolvimento

Este capítulo visa detalhar o processo de desenvolvimento e refatoração da API, apresentando a análise de requisitos, arquitetura utilizada em ambas versões e situações onde foram aplicados os *design patterns* e princípios SOLID, bem como a motivação por trás da implementação.

5.1 Levantamento de Requisitos

Para realizar o desenvolvimento do estudo de caso foi necessário documentar e planejar seus requisitos que segundo Sommerville (2011) representam as descrições do que o sistema deve fazer, os serviços que oferece e as restrições a seu funcionamento.

Os requisitos funcionais são declarações de serviços que o sistema deve fornecer (SOMMERVILLE, 2011). Sendo assim, analisando a Tabela 3 podemos perceber que o foco da API é realizar as operações CRUD com clientes, categorias, produtos, reposições, vendas, custos, usuários e permissões.

Tabela 3 – Requisitos Funcionais

Identificador	**Descrição**	**Prioridade**	**Depende de**
RF01	O Sistema deve permitir o registro, exclusão e atualização de clientes.	Alta	-
RF02	O Sistema deve permitir o registro, exclusão e atualização de categorias.	Alta	-
RF03	O Sistema deve permitir o registro, exclusão e atualização de produtos.	Alta	RF02
RF04	O Sistema deve permitir o registro, exclusão e atualização de reposições.	Alta	RF03
RF05	O Sistema deve permitir o registro, exclusão e atualização de vendas.	Alta	RF03, RF01 e RF07
RF06	O Sistema deve permitir o registro, exclusão e atualização de custos.	Média	-
RF07	O Sistema deve permitir o registro, exclusão e atualização de usuários.	Alta	-
RF08	O Sistema deve permitir o registro, exclusão e atualização de permissões.	Média	-
RF09	O Sistema deve possuir autenticação por meio de JWT.	Alta	RF07
RF10	O Sistema deve permitir a busca de clientes pelo nome.	Média	RF01
RF11	O Sistema deve permitir a busca de categorias pelo nome.	Média	RF02
RF12	O Sistema deve permitir a busca de produtos por código ou pelo nome da categoria.	Média	RF03
RF13	O Sistema deve permitir a busca de reposições pelo nome da categoria ou por código do produto.	Média	RF04
RF14	O Sistema deve permitir a busca de vendas pelos mesmos atributes de RF13 ou por nome do cliente.	Média	RF05

Fonte: Do autor, 2022

Para Sommerville (2011) os requisitos não funcionais representam as restrições aos serviços ou funções oferecidas pelo sistema. Visto que o estudo de caso será utilizado

apenas para a comparação e geração de métricas, os requisitos presentes na Tabela 4 são voltados em sua maioria para o desenvolvimento da aplicação.

Tabela 4 – Requisitos Não Funcionais

Identificador	**Descrição**	**Categoria**	**Escopo**	Prioridade	Depende de
RNF01	O sistema deve ser desenvolvido utilizando TypeScript.	Implementação	Desenvolvimento	Alta	-
RNF02	O Sistema deve ser agnóstico ao banco de dados.	Implementação	Desenvolvimento	Alta	-
RNF03	O Sistema deve ser desenvolvido utilizando os padrões SOLID.	Padrões	Desenvolvimento	Alta	-
RNF04	O Sistema deve ser desenvolvido utilizando padrões de projeto.	Padrões	Desenvolvimento	Alta	-
RNF05	O banco de dados deve permitir um alto volume de dados.	Disponibilidade	Infraestrutura	Médio	-

Fonte: Do autor, 2022

As regras de negócio são padrões que definem as diretrizes de funcionamento do projeto, logo, todas as atividades relacionadas ao sistema devem respeitá-las (SOMMERVILLE, 2011). Na Tabela 5 são apresentadas as regras definidas para o escopo do presente trabalho.

Tabela 5 – Regras de Negócio

Identificador	**Descrição**	Prioridade	Depende de
RN01	A quantidade de produtos só é alterada ao cadastrar uma reposição ou venda.	Alta	-
RN02	Os produtos deverão estar relacionados a uma categoria.	Alta	-
RN03	As vendas só poderão ser feitas caso tenha produtos em estoque.	Alta	-
RN04	Descontos só podem ser aplicados em vendas com pagamento via transferências ou dinheiro.	Alta	-
RN05	O registro de reposição só poderá ser feito caso o produto já esteja cadastrado.	Alta	-

Fonte: Do autor, 2022

5.2 Arquitetura

A arquitetura de um software visa detalhar tanto aspectos de baixo nível quanto a estrutura de alto nível do projeto, formando um tecido contínuo que define e molda o sistema (MARTIN et al., 2018).

Essa seção tem como foco discutir os módulos presentes na arquitetura do estudo de caso, apresentando sua responsabilidade, exemplos e de que forma foram interligados

para atingir o objetivo do trabalho.

5.2.1 Entidades

Para Martin et al. (2018) as entidades são responsáveis por representar um determinado conceito relevante ao sistema, por exemplo, em uma API para gerenciamento de estoque é comum ter entidades como: Produto, Venda e Cliente.

Essas entidades são a base de todo o fluxo de armazenamento, distribuição e alteração de dados, pois os métodos que interagem com o banco de dados necessitam de uma entidade para que possam realizar todas as operações necessárias. Os controladores e casos de uso também são dependentes das entidades, já que é neles que está contida a lógica de criação e distribuição das entidades ao repositório correto para que a operação desejada seja concluída.

O Código Fonte 1 apresenta um exemplo de entidade do sistema, nele é possível observar que a entidade é uma classe contendo atributos que juntos representam um Produto. A classe também pode ter métodos responsáveis por realizar alguma ação relacionada a entidade.

Analisando o Código Fonte 1, também pode-se perceber que o produto está diretamente relacionado a uma entidade de categoria, pois ele contém uma referência obrigatória a uma instância da mesma.

Para o desenvolvimento do estudo de caso, foi necessário mapear todas as entidades presentes no diagrama de classes no Apêndice A.1, bem como seus respectivos relacionamentos.

Código Fonte 1 – Entidade Produto

```
1  class Product {
2      public readonly id: string;
3      public amount: number;
4      public size: string;
5      public color: string;
6      public code: number;
7      public category: Category;
8
9      public costPrice: number;
10     public salePrice: number;
11
12     constructor(props: Omit<Product, "id">, id?: string) {
13         Object.assign(this, props);
14
15         this.size = this.size.toUpperCase();
16
17         if (!id) {
```

```
18              this.id = uuid();
19          } else {
20              this.id = id;
21          }
22      }
23  }
```

5.2.2 Repositórios

Com o intuito de centralizar a comunicação com o banco de dados foram criados repositórios, que oferecem funções responsáveis por realizar operações específicas na base de dados configurada.

Os repositórios utilizam como base a biblioteca Knex, que fornece uma instância de comunicação com a base de dados e possibilita a criação de *queries* SQL utilizando encadeamento de métodos que são agnósticos a especificidades de cada banco de dados, conforme comentado na subseção 2.1.2.

As operações de salvar, editar, remover e listar registros estão presentes em todos os repositórios, porém também foi preciso mapear os demais métodos contidos nas entidades do diagrama de classes no Apêndice A.1.

Código Fonte 2 – Método para Recuperar Cliente do Banco de Dados

```
1  findById = async () => {
2      const client = await this.connection(Tables.CLIENT).select<Client>("
           *").where({ id }).first();
3
4      if (!client) return;
5
6      return new Client(client, client.id);
7  }
```

5.2.3 Casos de Uso

As funcionalidades do sistema são representadas por casos de uso, que são responsáveis por controlar as entidades, aplicar as regras de negócio e garantir que o repositório correto seja utilizado para realizar as devidas alterações na base de dados.

Martin et al. (2018) afirma que “os casos de uso controlam a dança das entidades”, ou seja, controlam as regras que especificam quando e de que forma as entidades serão criadas e utilizadas para atingir os objetivos do sistema.

5.2.4 *Data Transfer Object*

O *Data Transfer Object* (DTO) é um objeto simples utilizado para transferir dados de um local a outro na aplicação sem conter lógicas de negócio (MARTIN, 2009). Geralmente, é utilizado para realizar a transferência de dados entre diferentes camadas do sistema que necessitam de dados complexos para realizar suas operações.

Sendo assim, o DTO sempre estará presente em operações de criação e edição de dados, pois os dados necessários para realizar as demais operações são simples e podem ser reduzidos a cadeias de caracteres e números, sem a necessidade de definir um DTO.

Código Fonte 3 – DTO para Criação de Cliente

```
interface CreateClientDTO {
  name: string;
  email?: string;
  cep?: string;
  cpf?: string;
  street?: string;
  number?: string;
  complement?: string;
  neighborhood?: string;
  state?: string;
  city?: string;
  phone?: string;
  birthday?: Date;
}
```

5.2.5 Controlador

Segundo Martin et al. (2018) os casos de uso devem receber dados comuns de entrada e retornar respostas simples. Esses dados não devem depender de nada, ou seja, não devem derivar de interfaces específicas de frameworks como *HttpRequest* ou *HttpResponse*.

Para cumprir as especificações supracitadas, foram criados controladores que atuam entre as rotas da aplicação e os casos de uso, utilizando DTOs para se comunicar e realizando o papel de interface com as funcionalidades específicas do framework Express.

Contudo, os controladores são responsáveis por validar se os dados recebidos estão de acordo com o esperado, enviá-los para o caso de uso correto e por fim, retornar uma resposta HTTP para identificar se tudo ocorreu conforme o esperado ou se ocorreu algum erro.

5.3 Implementação

A fim de apresentar os principais aspectos da implementação da primeira versão do estudo de caso, a presente seção detalha o fluxo de manutenção de usuários, que é o mesmo aplicado nas demais manutenções CRUD da API REST, porém respeitando as regras de negócio, detalhes e relacionamentos do Diagrama de Classes no Apêndice A.1.

5.3.1 Rotas

O ponto de acesso de uma API REST desenvolvida com Express são as rotas, que devem ser mapeadas a um verbo HTTP[1] e uma função que irá tratar a requisição retornando uma determinada resposta (EXPRESS, 2022).

Sendo assim, no Código Fonte 4 são apresentadas as rotas para manutenção de usuários. É possível notar que a classe *UserRoutes* implementa uma interface *Routes*, presente no Apêndice B.1, que define o formato que a classe deverá seguir, ou seja, implementar um método *apply* que retorna um objeto do tipo *Router* do Express. Esse método recebe por parâmetro uma coleção do tipo *Handler*, que conforme definido por Express (2022), são funções executadas em sequência para realizar operações na requisição até que uma resposta seja retornada ou que a próxima função da cadeia seja chamada[2].

Código Fonte 4 – Classe de Rotas do Usuário

```
1  import { createUserController } from "../../useCases/user/create";
2  import { showUserController } from "../../useCases/user/show";
3  import { showAllUsersController } from "../../useCases/user/showAll";
4  import { removeUserController } from "../../useCases/user/remove";
5  import { updateUserController } from "../../useCases/user/update";
6
7  export class UserRoutes implements Routes {
8    apply = (handlers: Handler[]) => {
9      const router = Router();
10
11     router.post("/users", ...handlers, createUserController.handle);
12     router.get("/users/:id", ...handlers, showUserController.handle);
13     router.get("/users", ...handlers, showAllUsersController.handle);
14     router.put("/users/:id", ...handlers, updateUserController.handle);
15     router.delete("/users/:id", ...handlers, removeUserController.handle
           );
16     return router;
17   };
18 }
```

[1] Verbo HTTP: Um conjunto de métodos de requisição responsáveis por indicar a ação a ser executada para um dado recurso (MOZILLA, 2022b).

[2] Essa cadeia de funções é denominada middleware e é uma implementação do padrão de projeto comportamental Chain of Responsibility (HO, 2022).

Desta forma, a classe *UserRoutes* está aplicando o SRP, pois sua única razão de mudar está relacionada a sua responsabilidade que é definir as rotas dos usuários. Além do DIP, já que quem define os *middlewares* que as rotas irão utilizar é o código cliente.

As rotas são definidas a partir de uma instância da função *Router*, que contém funções representando todos os verbos HTTP disponíveis, sendo necessário atribuir uma rota única para cada controlador, bem como encadear os *middlewares* recebidos por inversão de dependência. No Apêndice B.1 pode-se perceber que a classe também pode especificar se é pública através do atributo *noAuth*, que deve ser tratado pelo código cliente para fornecer ou não o middleware de autenticação.

5.3.2 Caso de Uso

Analisando as importações do Código Fonte 4, é possível notar que os controladores são importados da pasta *useCases*, que é onde estão definidos todos os casos de uso da aplicação. Cada caso de uso deve conter um controlador, um DTO opcional e a implementação concreta do caso de uso, além do código cliente que instância as classes, interligando repositórios, controladores e casos de uso.

No Código Fonte 5 é apresentado o controlador de criação de usuários, que recebe por meio do construtor uma instância da classe *CreateUserUseCase* do Código Fonte 6. O controlador então implementa o método *handle*, que por sua vez recebe dois parâmetros do tipo *Request* e *Response* que são interfaces específicas do framework Express. No corpo do método é feito um tratamento de erros utilizando blocos *Try/Catch*[3], onde o método *execute* do caso de uso recebido pelo construtor será executado com os dados recebidos no corpo da requisição, se nenhuma exceção for gerada, será retornada uma resposta com status HTTP 201[4].

Código Fonte 5 – Controlador de Criação de Usuário

```
1  class CreateUserController {
2    constructor(private createUserUseCase: CreateUserUseCase) {}
3
4    handle = async (request: Request, response: Response) => {
5      try {
6        await this.createUserUseCase.execute(request.body);
7
8        return response.sendStatus(201);
9      } catch (error) {
10       console.log(error);
11       const { message } = error;
```

[3] Try/Catch: Marcam um bloco de declarações para testar (try), e especifica uma resposta, caso uma exceção seja lançada (MOZILLA, 2022c).

[4] O status HTTP 201 é utilizado como resposta de sucesso, indica que a requisição foi bem sucedida e que um novo recurso foi criado (MOZILLA, 2022a).

```
12        if (error instanceof RequestError) {
13          const { status } = error;
14          return response.status(status).json({ message });
15        }
16        return response.status(500).json({ message });
17      }
18    };
19  }
```

Ao analisar a implementação concreta do caso de uso executado pelo controlador, percebe-se que o mesmo espera receber duas classes que implementam as interfaces *UserRepository* e *PermissionRepository*, apresentados respectivamente nos Apêndices B.2 e B.3. Segundo Martin et al. (2018) essa dependência por interfaces e não implementações concretas garante a aplicação do DIP, pois eventuais alterações feitas na implementação concreta não afetarão a classe dependente.

Código Fonte 6 – Caso de Uso de Criação de Usuário

```
1  class CreateUserUseCase {
2    constructor(private userRepository: UserRepository, private
       permissionRepository: PermissionRepository) {}
3
4    execute = async (data: CreateUserDTO) => {
5      const userPermissionsPromise = data.permissions.map((id) => {
6        return this.permissionRepository.findById(id);
7      });
8
9      const userPermissions = await Promise.all(userPermissionsPromise);
10     const user = new User({ ...data, permissions: userPermissions });
11
12     await this.userRepository.save(user);
13   };
14 }
```

Em seguida, é definido o método *execute*, que recebe um objeto do tipo *CreateUserDTO*, presente no Apêndice B.4, contendo o nome do usuário e um vetor de identificadores de cada permissão que são utilizados para realizar a criação de uma entidade *User*, definida no Apêndice B.8, que é passada por parâmetro para o método *save* do repositório de usuários recebido pelo construtor da classe.

Por fim, no código cliente apresentado no Código Fonte 7 são instanciadas as classes *CreateUserUseCase*, com implementações concretas de seus respectivos repositórios e *CreateUserController*, com a instância do caso de uso.

Código Fonte 7 – Código Cliente do Caso de Uso

```
1 const postgresPermissionsRepository = new PostgresPermissionsRepository(
    postgres);
2 const postgresUserRepository = new PostgresUserRepository(postgres,
    postgresPermissionsRepository);
3
4 const createUserUseCase = new CreateUserUseCase(postgresUserRepository,
    postgresPermissionsRepository);
5 const createUserController = new CreateUserController(createUserUseCase)
    ;
6
7 export { createUserUseCase, createUserController };
```

5.3.3 Repositório

Conforme apresentado na subseção 5.2.2, o repositório é responsável por realizar a comunicação com o banco de dados. Desta forma, o Código Fonte 8 contém as implementações dos métodos definidos pela interface *UserRepository*, presente no Apêndice B.2, que foi utilizada para aplicar o DIP no caso de uso e estabelecer um contrato no qual a classe *PostgresUserRepository* deve seguir para ser utilizada pelo restante da aplicação.

Analisando a classe *PostgresUserRepository*, é possível perceber que em seu construtor espera-se receber uma instância de uma classe que implemente a interface *PermissionRepository*, apresentada no Apêndice B.3, além de uma instância de conexão do Knex, que é utilizada para construir as *queries* do banco de dados configurado.

Na interface *UserRepository*, é possível notar que alguns métodos esperam receber por parâmetro uma entidade *User* ou retornar um objeto do mesmo tipo. Porém, no banco de dados, as permissões são armazenadas em outra tabela, sendo necessário realizar uma adaptação da entidade para o formato esperado pelas tabelas e vice versa. Para isso, foram criados os métodos *parseUserToDB* e *parseUserFromDB*, responsáveis por converter os dados de uma camada para outra, o que viola o SRP, pois a classe tem mais de uma responsabilidade.

Código Fonte 8 – Repositório de Usuário

```
1 class PostgresUserRepository implements UserRepository {
2   constructor(private connection: Knex, private permissionRepository:
      PermissionRepository) {}
3
4   private parseUserToDB = (user: User): UserDB => ({
5     id: user.id,
6     name: user.name,
7     password: user.password,
8   });
9
```

```
10    private parseUserFromDB = (user: UserDB, permissions: Permission[]):
          User => {
11      return new User({ ...user, permissions }, user.id);
12    };
13
14    ...
15  }
```

5.4 Refatoração

Para detalhar o processo de refatoração da primeira versão da API REST, essa seção está dividida em quatro subseções, onde cada uma apresentará a aplicação de um padrão de projeto, bem como os motivos pelo qual o mesmo foi implementado.

5.4.1 *Adapter*

O padrão de projeto *Adapter* foi implementado pelo fato de diferentes camadas da aplicação esperarem receber os mesmos dados, porém em formatos diferentes, ou seja, em alguns casos, existia uma incompatibilidade entre o formato das tabelas no banco de dados e as entidades mapeadas no sistema, sendo a mais comum o relacionamento entre tabelas.

Os *Adapters* relacionados a entidade de usuário estão presentes no Código Fonte 9, onde inicialmente foi definido o formato das tabelas de usuário e sua associativa com a tabela de permissões. Em seguida, foram criados três *Adapters*: *UserModelAdapter* que estende da entidade *User* e é responsável por adaptar os dados retornados pelo banco de dados para o formato de entidades, que são aceitas pelo restante da aplicação; *UserEntityAdapter* que estende da classe *UserModel* e deve converter os dados de uma entidade para que sejam aceitos pela tabela de usuários; *UserPermissionEntityAdapter* estende da classe *UserPermissionModel* e é utilizada para gerar um registro da tabela de relacionamento entre usuário e permissão.

A utilização dos *Adapters* pode ser observada em alguns métodos da nova versão da classe *PostgresUserRepository*, apresentados nos Apêndices C.7 e C.8, onde é possível notar que para converter os dados de uma camada para outra, basta instanciar o adaptador correto com os dados necessários para realizar a conversão.

Código Fonte 9 – Implementação do *Adapter*

```
class UserModelAdapter extends User {
  constructor(user: UserModel, permissions?: Permission[]) {
    super(
      {
        name: user.name,
        password: user.password,
        permissions,
      },
      user.id,
    );
  }
}

class UserEntityAdapter extends UserModel {
  constructor(user: User) {
    super();
    this.id = user.id;
    this.name = user.name;
    this.password = user.password;
  }
}
```

5.4.2 *Factory Method*

Observando o cliente do caso de uso da primeira versão presente no Código Fonte 7, percebe-se que o mesmo é responsável por instanciar todas as classes atreladas ao repositório, ou seja, se o caso de uso espera receber uma instância de *PostgresUserRepository*, o cliente também deverá instanciar a classe *PostgresPermissionRepository* para passa-lá ao construtor do repositório de usuários. Conforme a complexidade do repositório aumentar, ele precisará de mais repositórios para ser instanciado, e caso o mesmo seja utilizado por diferentes partes do código, todos deverão ser atualizados para lidar com as novas dependências.

Sendo assim, foi implementada uma classe responsável por apenas criar um repositório com todas as suas dependências e retornar uma instância do mesmo. Para isso, foi criada a classe abstrata *RepositoryFactory*, presente no Apêndice C.4, contendo o método abstrato *getRepository*, que será implementado pelas subclasses retornando a instância de um repositório.

No Código Fonte 10 está a implementação da fábrica *UserRepositoryFactory*, que é responsável por instanciar a classe *PostgresUserRepository*, bem como suas dependências. É possível notar que o método *getRepository* aceita uma instância de conexão do Knex para aplicar o DIP e não acoplar o repositório a uma única instância do banco de dados,

além disso, a fábrica está utilizando composição para obter o repositório de permissões, desta forma, o problema supracitado não se repetirá.

Código Fonte 10 – Aplicação do *Factory Method*

```
class UserRepositoryFactory extends RepositoryFactory<UserRepository> {
  getRepository(connection?: Knex): UserRepository {
    const permissionRepository = new PermissionRepositoryFactory();

    return new PostgresUserRepository(
      connection ?? Connection.getInstance(),
      permissionRepository.getRepository(connection),
    );
  }
}
```

5.4.3 *Template Method*

Ao analisar o Código Fonte 5, percebe-se que a maior parte do código presente na função não é específico da funcionalidade, pois, o mesmo código se repete em todos os demais controladores da aplicação. O único código que é relacionado a criação de usuários é a chamada para o caso de uso e o retorno de um *status code* 201, o restante é apenas um construtor que recebe um caso de uso e um tratamento genérico de erro.

Para resolver essa duplicação de código nos controladores, foi criada uma classe abstrata para servir como o *Template* da Figura 3, denominada *Controller*, que recebe em seu construtor um caso de uso e possui um método *handle* com o mesmo *Try/Catch*, onde é feita uma chamada para a função abstrata *action* que deve ser definida pela classe concreta que implementa esse contrato. A classe *Controller* é apresentada no Código Fonte 11.

Código Fonte 11 – Implementação do *Template*

```
abstract class Controller<U extends UseCase> {
  protected readonly useCase: U;

  constructor(useCase: U) {
    this.useCase = useCase;
  }

  public handle = async (request: Request, response: Response) => {
    try {
      return await this.action(request, response);
    } catch (error) {
      console.log(error);
      const { message } = error;
      if (error instanceof RequestError) {
```

```
15          const { status } = error;
16          return response.status(status).json({ message });
17        }
18        return response.status(500).json({ message });
19      }
20    };
21
22    protected abstract action: (request: Request, response: Response) =>
          Promise<Response> | void;
23  }
```

Também é possível notar na implementação concreta do controlador do Código Fonte 12 que o DIP foi aplicado, pois agora o construtor é definido na classe abstrata *Controller*, onde espera-se receber um caso de uso que implemente a interface *UseCase*, presente no Apêndice C.1, que aceita dois genéricos[5] utilizados para mapear o que o mesmo pode receber e deve retornar.

Código Fonte 12 – Aplicação do *Template Method*

```
1  class CreateController extends Controller<UseCase<DTO, void>> {
2    protected action = async (request: Request, response: Response) => {
3      await this.useCase.execute(request.body);
4      return response.sendStatus(201);
5    };
6  }
```

[5] Genéticos: Funcionalidade do TS que possibilita a criação de um componente que pode funcionar em vários tipos, em vez de em um único (TYPESCRIPT, 2022).

6 Resultados

Para apresentar os resultados obtidos com o refatoração da API REST, o presente capítulo está dividido em duas seções, na primeira foi feita uma análise dos principais ganhos obtidos em relação aos atributos de qualidade da Tabela 1 no que tange aos padrões de projeto, com base na teoria apresentada por Gamma et al. (1995) e percepções do autor quanto a utilização dos mesmos na refatoração do sistema.

Já na segunda seção, serão discutidas as métricas obtidas com a aplicação da ferramenta SonarQube, introduzida na subseção 2.4, em ambas versões do sistema e realizando uma comparação entre os principais aspectos apontados pela mesma.

6.1 Qualidade do Código

A utilização do padrão de projeto *Adapter* removeu do repositório a responsabilidade de converter os dados para o formato aceito pelas outras camadas do sistema. Desta forma, garantindo a aplicação do SRP, pois após a refatoração o repositório só tem uma única responsabilidade e razão para mudar que é a logica de interação com o banco de dados, tornando assim a classe mais enxuta e de fácil manutenção (MARTIN; NEWKIRK; KOSS, 2003).

Com a implementação do *design pattern Adapter*, também obtêm-se a aplicação do OCP, visto que é possível incluir novos adaptadores ao código sem a necessidade de alterar os que já estão presentes, além de possibilitar a reutilização do adaptador em outros locais do sistema.

O *Factory Method* simplificou a instanciação dos repositórios, pois após a refatoração, toda a lógica de criação e adição de dependências aos mesmos foi encapsulada em uma fábrica, que além de centralizar essa operação, trouxe estabilidade e manutenibilidade ao código, justamente pelo fato de que caso um repositório seja refatorado e suas dependências mudem, apenas a fábrica deverá sofrer alterações, desta forma, evita a obrigação de alterar todos os códigos cliente que utilizam o repositório ou eventuais erros causados pela instanciação incorreta da classe.

Conforme observado na Subseção 5.4.2 e descrito por Shvets (2021), a utilização do padrão *Factory Method* aplica o SRP ao mover o código de criação do repositório para um único local do programa e o OCP, pois novos tipos de repositório podem ser introduzidos ao sistema sem impactar no código cliente, além de evitar acoplamentos fortes entre o consumidor e a implementação concreta de classes. Esse resultado pode ser observado ao analisar a diferença entre os Códigos Fonte 13 e 14.

Código Fonte 13 – Código Cliente do Repositório de Vendas na Primeira Versão

```
1  const postgresCategoryRepository = new PostgresCategoryRepository(
       postgres);
2  const postgresClientRepository = new PostgresClientRepository(postgres);
3  const postgresProductRepository = new PostgresProductRepository(postgres
       , postgresCategoryRepository);
4  const postgresPurchaseProductRepository = new
       PostgresSaleProductRepository(
5    postgres,
6    postgresProductRepository,
7    postgresClientRepository,
8  );
9
10 const allSalesUseCase = new AllSalesUseCase(
       postgresPurchaseProductRepository);
11
12 const allSalesController = new AllPurchasesController(allSalesUseCase);
13
14 export { allSalesUseCase, allSalesController };
```

Código Fonte 14 – Código Cliente do Repositório de Vendas Utilizando o *Factory Method*

```
1  const saleRepositoryFactory = new SaleRepositoryFactory();
2
3  const listUseCase = new ListUseCase(saleRepositoryFactory.getRepository
       ());
4
5  const listController = new ListController(listUseCase);
6
7  export default { useCase: listUseCase, controller: listController };
```

Por fim, com a utilização do *Template Method*, foi possível reduzir a complexidade e número de linhas dos controladores, tornando-os mais simples e auxiliando na compreensão de seu funcionamento. Porém, Gamma et al. (1995) afirma que a implementação deste padrão tende a tornar-se mais complexa conforme o número de etapas aumenta e Shvets (2021) também explica que em alguns casos, o LSP pode ser violado ao suprimir uma etapa padrão de implementação através da subclasse. Os resultados podem ser observados analisando a diferença entre os Códigos Fonte 5 e 12.

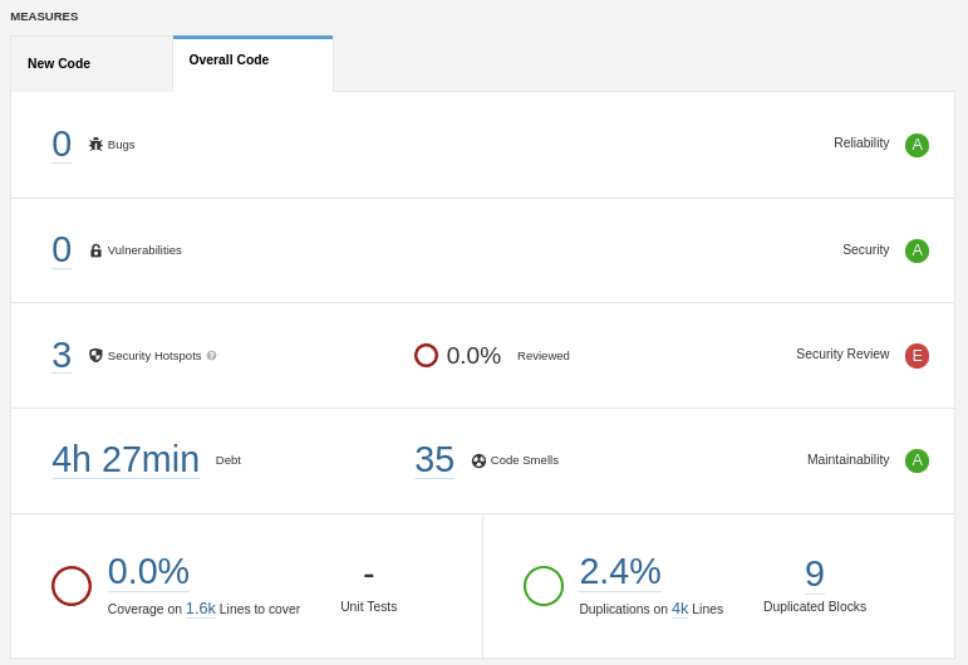
6.2 Métricas

Conforme apresentado na Seção 2.4, a ferramenta SonarQube é capaz de obter diferentes métricas relacionadas ao código de um sistema. Porém, para analisar os impactos causados pela utilização dos padrões de projeto, os aspectos de segurança e cobertura não

foram analisados, pois segundo Singh e Gautam (2016), não são atributos impactados por *design patterns*.

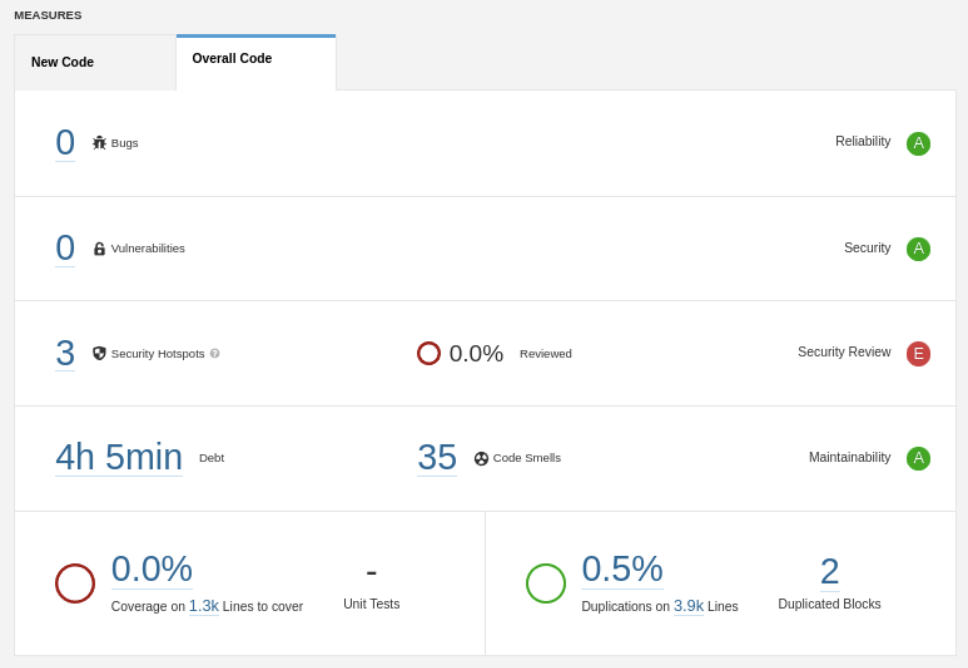
Nas Figuras 5 e 6, são apresentadas as visões gerais do sistema em suas duas versões. Nelas é possível perceber que a quantidade de *bugs* identificadas pela ferramenta foi de 0 em ambas as versões, resultando em uma nota A de confiabilidade. Outro ponto em comum foi a manutenibilidade, pois o número de *code smells* encontrado é o mesmo, o que gerou um débito técnico parecido e uma nota A para as duas versões. A única métrica com uma diferença significativa na *Dashboard* foi a porcentagem de linhas duplicadas, sendo 2,4% na primeira versão e 0,5% na segunda.

Figura 5 – Overview SonarQube Primeira Versão



Fonte: Do autor, 2022

Figura 6 – Overview SonarQube Segunda Versão



Fonte: Do autor, 2022

A duplicação de código pode ser visualizada de forma mais detalhada nas Figuras 7 e 8, onde é possível notar que a ferramenta identificou na primeira versão do sistema um total de 127 linhas duplicadas no diretório de casos de uso, o que representa 3,5% do total de linhas dessa pasta. Já na análise da segunda versão presente na Figura 8, foram identificadas 24 linhas duplicadas nos casos de uso, representando 0,8% do total de linhas do diretório.

Figura 7 – Duplicação de Código na Primeira Versão

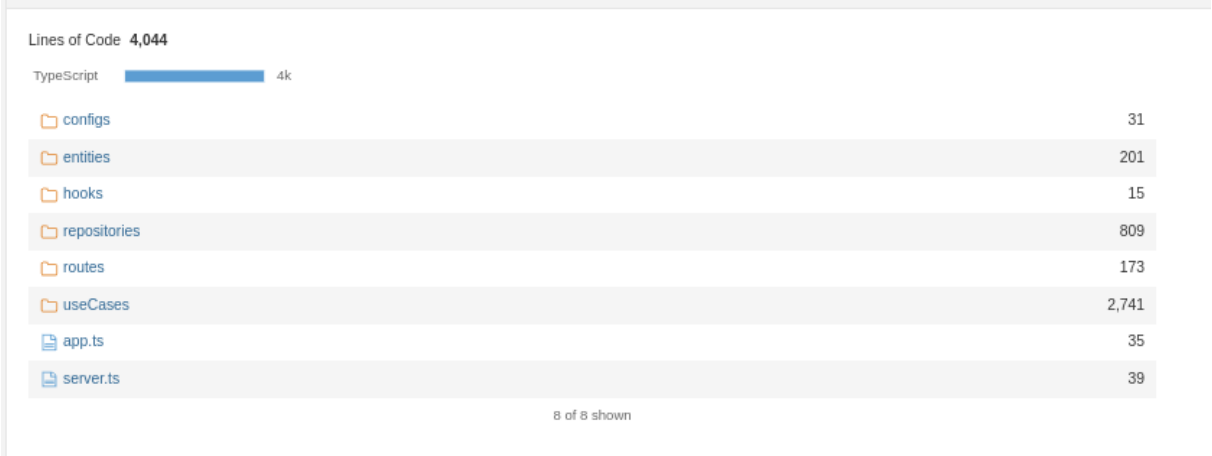
Fonte: Do autor, 2022

Figura 8 – Duplicação de Código na Segunda Versão

Fonte: Do autor, 2022

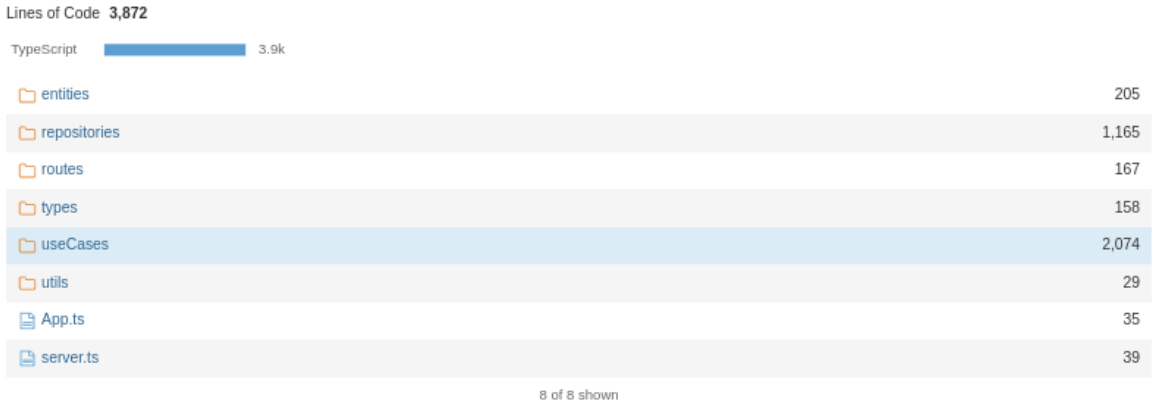
No quesito de quantidade de linhas de código, a principal diferença entre as duas versões está nos diretórios de casos de uso, onde o número foi reduzido de 2741 para 2074, e repositórios, cujo número aumentou de 809 para 1165. Porém, analisando a quantidade total de linhas nas Figuras 9 e 10, é possível perceber que o número foi reduzido em 172 linhas.

Figura 9 – Quantidade de Linhas na Primeira Versão



Fonte: Do autor, 2022

Figura 10 – Quantidade de Linhas na Segunda Versão



Fonte: Do autor, 2022

O número de linhas reduzidas nos casos de uso está diretamente relacionado a aplicação do *Template Method*, pois após a refatoração, os controladores se tornaram mais enxutos, visto que o processo de tratamento de erro foi movido para a classe *Template*. Já o aumento no número de linhas do diretório de repositórios se deu pela implementação de *Adapters*, que segundo Shvets (2021) é o comportamento esperado, já que novas subclasses serão introduzidas ao sistema. O padrão de projeto *Factory Method* impactou em ambos os casos, pois foi necessário adicionar novas classes responsáveis por fabricar instâncias de repositórios, que causaram o aumento no número de linhas desse diretório, porém reduziu dos casos de uso, visto que o código cliente não precisa lidar com as dependências do repositório.

Com relação a complexidade cognitiva, a ferramenta identificou uma redução de 142 pontos, o que significa que o sistema tornou-se mais compreensivo no geral. Porém, analisando a complexidade da pasta de repositórios nas Figuras 11 e 12, percebe-se que esse valor aumentou em 20 pontos, que é o comportamento esperado de acordo com a Tabela 2.

Figura 11 – Complexidade Cognitiva na Primeira Versão

Fonte: Do autor, 2022

Figura 12 – Complexidade Cognitiva na Segunda Versão

Fonte: Do autor, 2022

Ao analisar o diretório de casos de uso, a ferramenta aponta que a complexidade diminuiu em 165 pontos, porém, na Tabela 2, o resultado esperado é que a compreensibilidade ao aplicar o padrão *Template Method* seja afetada negativamente. Essa divergência ocorreu pelo fato de que o cenário onde o *design pattern* foi aplicado é relativamente simples e envolve apenas a implementação de um método abstrato e conforme esse número aumente, Gamma et al. (1995) afirmam que a complexidade das classes concretas tendem a aumentar, tornando o código menos compreensivo.

7 Considerações finais

Tendo em vista que os padrões de projeto apresentados por Gamma et al. (1995) estão sendo utilizados constantemente para o desenvolvimento e solução de problemas e paradigmas relacionados a software, bem como *frameworks* modernos como o Express (HO, 2022), o presente trabalho teve como propósito a identificação dos impactos causados pela aplicação de alguns padrões na qualidade do código de uma API REST que realiza operações de controle de estoque.

De modo geral, notou-se que a utilização de *design patterns* impactou positivamente nos quesitos de qualidade e manutenibilidade do código, além de facilitar a aplicação de alguns princípios SOLID. Porém, é importante frisar que os resultados estão diretamente relacionados aos padrões de projeto utilizados no decorrer do trabalho.

Sendo assim, também foi possível notar que as métricas obtidas com a aplicação da ferramenta SonarQube foram positivas, pois a avaliação da segunda versão do sistema se mostrou superior a primeira no quesito de complexidade e duplicação de código, quando nos demais atributos se manteve sem alteração.

Contudo, vale ressaltar que as métricas obtidos podem variar de acordo com a escala do projeto no qual os padrões são aplicados, pois o número de linhas do sistema desenvolvido foi reduzido de 4044 para 3872 na segunda versão, porém, essa redução poderia ser mais significativa caso o número de linhas da primeira versão fosse maior.

Outro fator delimitador são as vivências e experiências de quem está aplicando os padrões de projeto, visto que seu propósito é disponibilizar um caminho com sugestões para auxiliar na resolução de determinados problemas, desta forma, cabe ao desenvolvedor seguir essas instruções de forma coerente para que os resultados estejam de acordo com o esperado.

Além destes aspectos, a capacidade de reconhecer o problema e identificar qual padrão de projeto é ideal para resolvê-lo também impacta nos resultados, pois, caso os padrões sejam utilizados de forma incorreta, tendem a causar impactos negativos e acabam gerando novos problemas no futuro, conforme apresentado por Khomh e Guéhéneuc (2008).

Em trabalhos futuros, propõem-se a utilização de outras ferramentas, como *Codacy*[1] ou *Embold*[2] para realizar a analise estática do código e identificar as métricas de qualidade de software, além da implementação dos demais padrões de projeto em diferentes casos de uso para avaliar seus respectivos impactos. Outro quesito que pode ser avaliado são as consequências geradas pela aplicação errônea de *design patterns*, para que sejam elencados

[1] https://www.codacy.com/
[2] https://embold.io/

também os riscos envolvidos na utilização incorreta dos mesmos.

Referências

ALEXANDER, C. *A pattern language: towns, buildings, construction*. [S.l.]: Oxford university press, 1977. Citado 2 vezes nas páginas 11 e 17.

AYLAN. *Duck Typing com Python*. 2019. Disponível em: <https://www.devmedia.com.br/duck-typing-com-python/40223>. Acesso em: 25 mai 2022. Citado na página 16.

BIERMAN, G.; ABADI, M.; TORGERSEN, M. Understanding typescript. In: SPRINGER. *European Conference on Object-Oriented Programming*. [S.l.], 2014. p. 257–281. Citado na página 14.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, IEEE, v. 20, n. 6, p. 476–493, 1994. Citado na página 22.

EXPRESS, O. F. *Framework web rápido, flexível e minimalista para Node.js*. 2022. Disponível em: <https://expressjs.com/pt-br/>. Acesso em: 25 mai 2022. Citado 2 vezes nas páginas 15 e 32.

FERNANDES, J. H. C. Qual a prática do desenvolvimento de software? *Ciência e Cultura*, Sociedade Brasileira para o Progresso da Ciência, v. 55, n. 2, p. 29–33, 2003. Citado na página 11.

FOWLER, M. *Refactoring: improving the design of existing code*. [S.l.]: Addison-Wesley Professional, 2018. Citado na página 20.

GAMMA, E. et al. *Design patterns: elements of reusable object-oriented software*. [S.l.]: Pearson Deutschland GmbH, 1995. Citado 11 vezes nas páginas 11, 17, 18, 19, 20, 24, 26, 40, 41, 46 e 47.

HO, C. *Design Patterns in Express.js*. 2022. Disponível em: <https://dzone.com/articles/design-patterns-in-expressjs>. Acesso em: 18 jun 2022. Citado 2 vezes nas páginas 32 e 47.

JORGENSEN, P. C. *Software testing: a craftsman's approach*. [S.l.]: Auerbach Publications, 2013. Citado na página 22.

KHOMH, F.; GUÉHÉNEUC, Y.-G. Do design patterns impact software quality positively? In: IEEE. *2008 12th European conference on software maintenance and reengineering*. [S.l.], 2008. p. 274–278. Citado 5 vezes nas páginas 8, 23, 24, 25 e 47.

KNEX. *SQL query builder*. 2022. Disponível em: <http://knexjs.org/>. Acesso em: 27 mai 2022. Citado na página 14.

MARDAN, A. *Express. js Guide: The Comprehensive Book on Express. js*. [S.l.]: Azat Mardan, 2014. Citado na página 15.

MARTIN, R. C. *Clean code: a handbook of agile software craftsmanship*. [S.l.]: Pearson Education, 2009. Citado na página 31.

MARTIN, R. C. et al. *Clean architecture: a craftsman's guide to software structure and design*. [S.l.]: Prentice Hall, 2018. Citado 8 vezes nas páginas 15, 16, 26, 28, 29, 30, 31 e 34.

MARTIN, R. C.; NEWKIRK, J.; KOSS, R. S. *Agile software development: principles, patterns, and practices*. [S.l.]: Prentice Hall Upper Saddle River, NJ, 2003. v. 2. Citado 5 vezes nas páginas 11, 15, 16, 22 e 40.

MCCABE, T. J. A complexity measure. *IEEE Transactions on software Engineering*, IEEE, n. 4, p. 308–320, 1976. Citado na página 22.

MOZILLA. *201 Created*. 2022. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status/201>. Acesso em: 18 jun 2022. Citado na página 33.

MOZILLA. *Métodos de requisição HTTP*. 2022. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods>. Acesso em: 17 jun 2022. Citado na página 32.

MOZILLA. *try...catch*. 2022. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/try...catch>. Acesso em: 18 jun 2022. Citado na página 33.

PAIXAO, J. *O que é SOLID: O guia completo para você entender os 5 princípios da POO*. 2019. Disponível em: <https://medium.com/desenvolvendo-com-paixao/o-que-%C3%A9-solid-o-guia-completo-para-voc%C3%AA-entender-os-5-princ%C3%ADpios-da-poo-2b937b3fc530>. Acesso em: 27 mai 2022. Citado na página 11.

REACT. *Uma biblioteca JavaScript para criar interfaces de usuário*. 2022. Disponível em: <https://pt-br.reactjs.org/>. Acesso em: 9 jun 2022. Citado na página 22.

SHVETS, A. *Dive Into DESIGN PATTERNS*. 2021. Disponível em: <https://refactoring.guru/design-patterns/book>. Acesso em: 25 mai 2022. Citado 9 vezes nas páginas 11, 15, 17, 18, 19, 26, 40, 41 e 45.

SINGH, B.; GAUTAM, S. The impact of software development process on software quality: a review. In: IEEE. *2016 8th international conference on computational intelligence and communication networks (CICN)*. [S.l.], 2016. p. 666–672. Citado 4 vezes nas páginas 8, 21, 23 e 42.

SOMMERVILLE, I. *Software Engineering, 9/E*. [S.l.]: Pearson Education Brasil, 2011. Citado 2 vezes nas páginas 27 e 28.

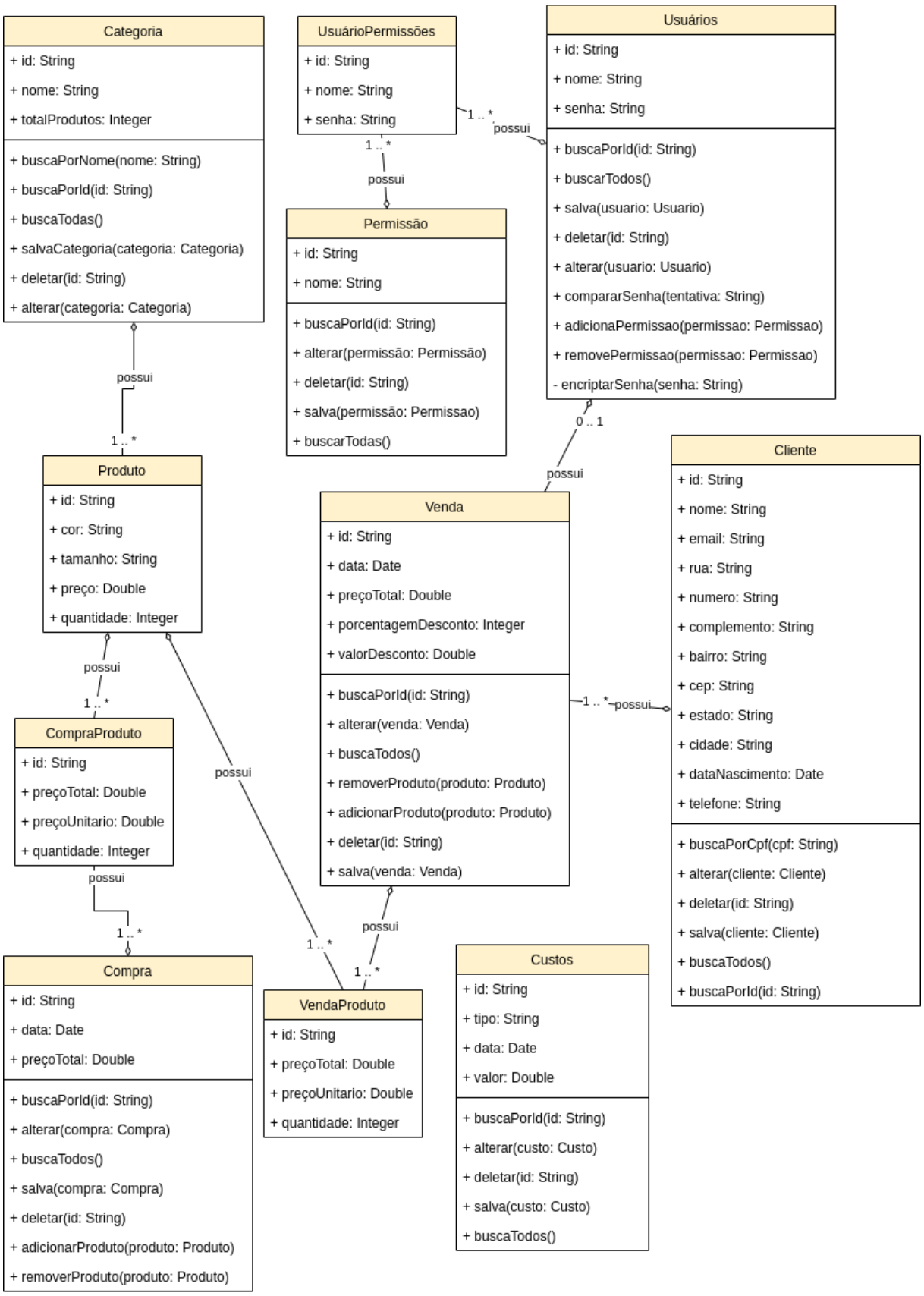
SONARSOURCE. *Your teammate for Code Quality and Code Security*. 2022. Disponível em: <https://www.sonarqube.org/>. Acesso em: 16 jun 2022. Citado na página 20.

TRODIN, M. *Applying the SOLID principles to JavaScript's React library*. 2021. Citado 2 vezes nas páginas 8 e 22.

TYPESCRIPT, M. *TypeScript is JavaScript with syntax for types*. 2022. Disponível em: <https://www.typescriptlang.org/>. Acesso em: 25 mai 2022. Citado 2 vezes nas páginas 14 e 39.

A Apêndice - Levantamento de Requisitos

A.1 Diagrama de Classes



B Apêndice - Implementação

B.1 Interface Routes

```typescript
import { Router, Handler } from "express";

export interface Routes {
  noAuth?: boolean;

  apply(handlers: Handler[]): Router;
}
```

B.2 Interface UserRepository

```typescript
import { User } from "../entities/User";

export interface UserRepository {
  findById(id: string): Promise<User | undefined>;
  getAll(): Promise<User[]>;
  save(user: User): Promise<void>;
  delete(id: string): Promise<void>;
  update(user: User, permissions?: string[]): Promise<void>;
  findByName(name: string): Promise<User | undefined>;
}
```

B.3 Interface PermissionRepository

```typescript
import { Permission } from "../entities/Permission";

export interface PermissionRepository {
  findById(id: string): Promise<Permission>;
  getAll(): Promise<Permission[]>;
  save(permission: Permission): Promise<void>;
  delete(id: string): Promise<void>;
  update(permission: Permission): Promise<void>;
}
```

B.4 Interface CreateUserDTO

```typescript
export interface CreateUserDTO {
  name: string;
```

```
  permissions: string[];
}
```

B.5 Classe UserRoutes

```
import { Routes } from "../Routes";
import { Handler, Router } from "express";
import { createUserController } from "../../useCases/user/create";
import { showUserController } from "../../useCases/user/show";
import { showAllUsersController } from "../../useCases/user/showAll";
import { removeUserController } from "../../useCases/user/remove";
import { updateUserController } from "../../useCases/user/update";

export class UserRoutes implements Routes {
  apply = (handlers: Handler[]) => {
    const router = Router();

    router.post("/users", ...handlers, createUserController.handle);

    router.get("/users/:id", ...handlers, showUserController.handle);

    router.get("/users", ...handlers, showAllUsersController.handle);

    router.put("/users/:id", ...handlers, updateUserController.handle);

    router.delete("/users/:id", ...handlers, removeUserController.handle
        );

    return router;
  };
}
```

B.6 Classe CreateUserController

```
import { Request, Response } from "express";
import { RequestError } from "../../../configs/error";
import { CreateUserUseCase } from "./CreateUserUseCase";

export class CreateUserController {
  constructor(private createUserUseCase: CreateUserUseCase) {}

  handle = async (request: Request, response: Response) => {
    try {
      await this.createUserUseCase.execute(request.body);

```

```
12        return response.sendStatus(201);
13      } catch (error) {
14        console.log(error);
15        const { message } = error;
16        if (error instanceof RequestError) {
17          const { status } = error;
18          return response.status(status).json({ message });
19        }
20        return response.status(500).json({ message });
21      }
22    };
23  }
```

B.7 Classe CreateUserUseCase

```
1  import { User } from "../../../entities/User";
2  import { PermissionRepository } from "../../../repositories/
       PermissionRepository";
3  import { UserRepository } from "../../../repositories/UserRepository";
4  import { CreateUserDTO } from "./CreateUserDTO";
5
6  export class CreateUserUseCase {
7    constructor(private userRepository: UserRepository, private
         permissionRepository: PermissionRepository) {}
8
9    execute = async (data: CreateUserDTO) => {
10     const userPermissionsPromise = data.permissions.map((id) => {
11       return this.permissionRepository.findById(id);
12     });
13
14     const userPermissions = await Promise.all(userPermissionsPromise);
15
16     const user = new User({ ...data, permissions: userPermissions });
17
18     await this.userRepository.save(user);
19   };
20 }
```

B.8 Classe User

```
1  import { v4 as uuid } from "uuid";
2  import crypto from "crypto";
3  import { Permission } from "./Permission";
4
5  export class User {
```

```
6   public readonly id: string;
7   public name: string;
8   public password?: string;
9   public permissions: Permission[];
10
11  constructor(props: Omit<User, "id" | "comparePassword">, id?: string,
        newPassword?: string) {
12    Object.assign(this, props);
13
14    if (!id) {
15      this.id = uuid();
16    } else {
17      this.id = id;
18    }
19
20    if (newPassword) {
21      this.password = this.encryptPassword(newPassword);
22    }
23  }
24
25  private encryptPassword = (password: string) => {
26    const salt = crypto.randomBytes(8).toString("hex");
27    const hashed = crypto.scryptSync(password, salt, 64);
28    return `${hashed.toString("hex")}.${salt}`;
29  };
30
31  public comparePassword = (attempt: string) => {
32    const [hashed, salt] = this.password.split(".");
33    const hashedAttempt = crypto.scryptSync(attempt, salt, 64);
34
35    return hashed === hashedAttempt.toString("hex");
36  };
37 }
```

B.9 Classe PostgresUserRepository

```
1  import { v4 as uuid } from "uuid";
2  import Knex from "knex";
3  import { User } from "../../entities/User";
4  import { Tables } from "../Tables";
5  import { UserRepository } from "../UserRepository";
6  import { Permission } from "../../entities/Permission";
7  import { PermissionRepository } from "../PermissionRepository";
8
9  interface UserDB {
10   id: string;
```

```
11    name: string;
12    password: string;
13  }
14
15  export class PostgresUserRepository implements UserRepository {
16    constructor(private connection: Knex, private permissionRepository:
          PermissionRepository) {}
17
18    private parseUserToDB = (user: User): UserDB => ({
19      id: user.id,
20      name: user.name,
21      password: user.password,
22    });
23
24    private parseUserFromDB = (user: UserDB, permissions: Permission[]):
          User => {
25      return new User({ ...user, permissions }, user.id);
26    };
27
28    save = async (u: User) => {
29      const trx = await this.connection.transaction();
30      await trx(Tables.USERS).insert(this.parseUserToDB(u));
31
32      const userPermissionsToInset = u.permissions.map((p) => ({
33        id: uuid(),
34        permission_id: p.id,
35        user_id: u.id,
36      }));
37
38      await trx(Tables.USER_PERMISSIONS).insert(userPermissionsToInset);
39
40      await trx.commit();
41    };
42
43    delete = async (id: string) => {
44      const trx = await this.connection.transaction();
45      await trx(Tables.USERS).where({ id }).del();
46      await trx(Tables.USER_PERMISSIONS).where({ user_id: id }).del();
47
48      await trx.commit();
49    };
50
51    findById = async (id: string) => {
52      const userDB = await this.connection(Tables.USERS).select("*").where
            ({ id }).first();
53
54      const permissionsDB = await this.connection(Tables.USER_PERMISSIONS)
```

```
      .select("*").where({ user_id: id });
55
56    const userPermissions: Permission[] = [];
57
58    for (const pDB of permissionsDB) {
59      const permission = await this.permissionRepository.findById(pDB.
          permission_id);
60
61      userPermissions.push(permission);
62    }
63
64    return this.parseUserFromDB(userDB, userPermissions);
65  };
66
67  getAll = async () => {
68    const users = await this.connection(Tables.USERS).select<UserDB[]>("
        *");
69
70    return users.map((u) => this.parseUserFromDB(u, []));
71  };
72
73  update = async (u: User, permissions?: string[]) => {
74    const trx = await this.connection.transaction();
75
76    const userToUpdate = this.parseUserToDB(u);
77    await trx(Tables.USERS).update(userToUpdate).where({ id: u.id });
78
79    if (permissions) {
80      await trx(Tables.USER_PERMISSIONS).where({ user_id: u.id }).del();
81
82      const userPermissionsToInset = permissions.map((permissionId) =>
          ({
83        id: uuid(),
84        permission_id: permissionId,
85        user_id: u.id,
86      }));
87
88      await trx(Tables.USER_PERMISSIONS).insert(userPermissionsToInset);
89    }
90
91    await trx.commit();
92  };
93
94  findByName = async (name: string) => {
95    const userDB = await this.connection(Tables.USERS).select<UserDB>("*
        ").where({ name }).first();
96
```

```
97      if (!userDB) return;
98
99      const permissionsDB = await this.connection(Tables.USER_PERMISSIONS)
            .select("*").where({ user_id: userDB.id });
100
101     const userPermissions: Permission[] = [];
102
103     for (const pDB of permissionsDB) {
104       const permission = await this.permissionRepository.findById(pDB.
              permission_id);
105
106       userPermissions.push(permission);
107     }
108
109     return this.parseUserFromDB(userDB, userPermissions);
110   };
111 }
```

B.10 Código Cliente Caso de Uso

```
1  import { postgres } from "../../../configs/knex";
2  import { PostgresPermissionsRepository } from "../../../repositories/
       implementations/PostgresPermissionRepository";
3  import { PostgresUserRepository } from "../../../repositories/
       implementations/PostgresUserRepository";
4  import { CreateUserController } from "./CreateUserController";
5  import { CreateUserUseCase } from "./CreateUserUseCase";
6
7  const postgresPermissionsRepository = new PostgresPermissionsRepository(
       postgres);
8  const postgresUserRepository = new PostgresUserRepository(postgres,
       postgresPermissionsRepository);
9
10 const createUserUseCase = new CreateUserUseCase(postgresUserRepository,
       postgresPermissionsRepository);
11 const createUserController = new CreateUserController(createUserUseCase)
       ;
12
13 export { createUserUseCase, createUserController };
```

B.11 Código Cliente Caso de Uso Listagem de Vendas

```
1  import { postgres } from "../../../configs/knex";
2  import { PostgresCategoryRepository } from "../../../repositories/
       implementations/PostgresCategoryRepository";
```

```
3  import { PostgresClientRepository } from "../../../repositories/
       implementations/PostgresClientRepository";
4  import { PostgresProductRepository } from "../../../repositories/
       implementations/PostgresProductRepository";
5  import { PostgresSaleProductRepository } from "../../../repositories/
       implementations/PostgresSaleProductRepository";
6  import { AllPurchasesController } from "./AllSalesController";
7  import { AllSalesUseCase } from "./AllSalesUseCase";
8
9  const postgresCategoryRepository = new PostgresCategoryRepository(
       postgres);
10 const postgresClientRepository = new PostgresClientRepository(postgres);
11 const postgresProductRepository = new PostgresProductRepository(postgres
       , postgresCategoryRepository);
12 const postgresPurchaseProductRepository = new
       PostgresSaleProductRepository(
13   postgres,
14   postgresProductRepository,
15   postgresClientRepository,
16 );
17
18 const allSalesUseCase = new AllSalesUseCase(
       postgresPurchaseProductRepository);
19
20 const allSalesController = new AllPurchasesController(allSalesUseCase);
21
22 export { allSalesUseCase, allSalesController };
```

C Apêndice - Refatoração

C.1 Interface UseCase

```
export interface UseCase<Params = any, Data = any> {
  execute: (p: Params) => Promise<Data>;
}
```

C.2 Classe Controller

```
import { RequestError } from "../utils/RequestError";
import { Request, Response } from "express";
import { UseCase } from "./UseCase";

export abstract class Controller<U extends UseCase> {
  protected readonly useCase: U;

  constructor(useCase: U) {
    this.useCase = useCase;
  }

  public handle = async (request: Request, response: Response) => {
    try {
      return await this.action(request, response);
    } catch (error) {
      console.log(error);
      const { message } = error;
      if (error instanceof RequestError) {
        const { status } = error;
        return response.status(status).json({ message });
      }
      return response.status(500).json({ message });
    }
  };

  protected abstract action: (request: Request, response: Response) =>
     Promise<Response> | void;
}
```

C.3 Classe CreateController

```
import { Request, Response } from "express";
```

```
import { UseCase } from "../../../types/UseCase";
import { Controller } from "../../../types/Controller";
import { DTO } from "./DTO";

export class CreateController extends Controller<UseCase<DTO, void>> {
  protected action = async (request: Request, response: Response) => {
    await this.useCase.execute(request.body);
    return response.sendStatus(201);
  };
}
```

C.4 Classe RepositoryFactory

```
import Knex from "knex";

export abstract class RepositoryFactory<R> {
  abstract getRepository(connection?: Knex): R;
}
```

C.5 Classe UserRepositoryFactory

```
import Knex from "knex";
import { RepositoryFactory } from "../../types/RepositoryFactory";
import { Connection } from "../Connection";
import { UserRepository } from "../../types/repositories/UserRepository"
    ;
import { PostgresUserRepository } from "./Repository";
import { PermissionRepositoryFactory } from "../PermissionRepository/
    Factory";

export class UserRepositoryFactory extends RepositoryFactory<
    UserRepository> {
  getRepository(connection?: Knex): UserRepository {
    const permissionRepository = new PermissionRepositoryFactory();

    return new PostgresUserRepository(
      connection ?? Connection.getInstance(),
      permissionRepository.getRepository(connection),
    );
  }
}
```

C.6 User Adapters

```
import { Permission } from "../../entities/Permission";
import { User } from "../../entities/User";
import { v4 as uuid } from "uuid";

export class UserModel {
  id: string;
  name: string;
  password?: string;
}

export class UserPermissionModel {
  id: string;
  permission_id: string;
  user_id: string;
}

export class UserModelAdapter extends User {
  constructor(user: UserModel, permissions?: Permission[]) {
    super(
      {
        name: user.name,
        password: user.password,
        permissions,
      },
      user.id,
    );
  }
}

export class UserEntityAdapter extends UserModel {
  constructor(user: User) {
    super();
    this.id = user.id;
    this.name = user.name;
    this.password = user.password;
  }
}

export class UserPermissionEntityAdapter extends UserPermissionModel {
  constructor(permission: Permission, user: User) {
    super();
    this.id = uuid();
    this.permission_id = permission.id;
    this.user_id = user.id;
  }
}
```

C.7 Método Save do Repositório PostgresUserRepository

```
1  save = async (u: User) => {
2    const trx = await this.connection.transaction();
3    const user = new UserEntityAdapter(u);
4
5    await trx("users").insert(user);
6
7    const userPermissionsToInset = u.permissions.map((p) => new
       UserPermissionEntityAdapter(p, u));
8
9    await trx("user_permissions").insert(userPermissionsToInset);
10
11   await trx.commit();
12 };
```

C.8 Método findById do Repositório PostgresUserRepository

```
1  findById = async (id: string) => {
2    const user = await this.connection("users").select<UserModel>("*").
       where({ id }).first();
3
4    if (!user) return;
5
6    const permissions = await this.permissionRepository.findByUserId(
       user.id);
7
8    return new UserModelAdapter(user, permissions);
9  };
```

C.9 Código Cliente Caso de Uso Listagem de Vendas

```
1  import { ListController } from "./Controller";
2  import { ListUseCase } from "./UseCase";
3  import { SaleRepositoryFactory } from "../../../repositories/
     SaleRepository/Factory";
4
5  const saleRepositoryFactory = new SaleRepositoryFactory();
6
7  const listUseCase = new ListUseCase(saleRepositoryFactory.getRepository
     ());
8
9  const listController = new ListController(listUseCase);
10
11 export default { useCase: listUseCase, controller: listController };
```