UNIVERSIDADE FEDERAL FLUMINENSE

CAIO CESAR FERNANDES TORRES

DESENVOLVIMENTO DE APLICAÇÕES WEB COM *SOLID* E *CLEAN ARCHITECTURE*

Niterói

2021

CAIO CESAR FERNANDES TORRES

DESENVOLVIMENTO DE APLICAÇÕES WEB COM *SOLID* E *CLEAN ARCHITECTURE*

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Sistemas de Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Computação.

Orientador(a):
Rafael Burlamaqui Amaral

NITERÓI
2021

Folha reservada para a ficha catalográfica

CAIO CESAR FERNANDES TORRES

DESENVOLVIMENTO DE APLICAÇÕES WEB COM SOLID E CLEAN ARCHITECTURE

Trabalho de Conclusão de Curso submetido ao Curso de Tecnologia em Sistemas de Computação da Universidade Federal Fluminense como requisito parcial para obtenção do título de Tecnólogo em Sistemas de Computação.

Niterói, ___ de _____ de ANO.

Banca Examinadora (provisório):

_____

Prof. ou Profª. <NOME>, <Título>. – Orientador ou Avaliador

<Sigla da Universidade> - <Nome da Universidade>

_____

Prof. ou Profª. <NOME>, <Título>. – Orientador ou Avaliador

<Sigla da Universidade> - <Nome da Universidade>

AGRADECIMENTOS

À minha mãe, Simone, pois sempre acreditou e confiou em mim.

Ao meu padrasto, Fernando, por sempre me lembrar da importância do estudo.

Ao meu pai, Walcknaer, por investir em minha educação e em minha jornada.

RESUMO

Boas práticas como os princípios *SOLID* e a *Clean Architecture* foram pensadas com o propósito de reduzir o esforço nas tarefas de manutenção de componentes de código ao se atentar a determinados detalhes durante a criação dos mesmos. O objetivo deste trabalho é estudar, explicar e implementar estas práticas em uma aplicação web e, posteriormente, refletir sobre os resultados alcançados com as mesmas.

Palavras-chaves: Boas Práticas, Código Limpo, Programação Orientada a Objetos

ABSTRACT

Good practices such as SOLID and Clean Architecture principles were designed with the purpose of reducing the effort in the maintenance tasks of code components by paying attention to certain details during their creation. The objective of this paper is to study, explain and implement these practices in a web application and, later, reflect on the results achieved with them.

Key words: Good Practices, Clean Code, Object-Oriented Programming.

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

DRY – *Don’t Repeat Yourself*
DDD – *Domain Driven Design*
TDD – *Test Driven Development*
KISS – *Keep It Simple, Stupid*
SRP – *Single Responsiblity Principle*
OCP – *Open-Closed Principle*
LSP – *Liskov Substitution Principle*
ISP – *Interface Segregation Principle*
DIP – *Dependency Inversion Principle*
ORM – *Object-Relational Mapping*
UI – *User Interface*
UML – *Unified Modeling Language*
HTTP – *Hyper Text Transfer Protocol*
API – *Application Programming Interface*

SUMÁRIO

1 INTRODUÇÃO

No desenvolvimento de aplicações corporativas ou com uma longa vida, a tendência é que o software sofra constantemente alterações a nível conceitual (por exemplo, um fluxo alternativo a fim de atender uma parceria nova ou a implementação do Pix como uma nova forma de pagamento) ou alterações a nível de infraestrutura (por exemplo, a migração de um servidor local para computação em nuvem ou o desmembramento da aplicação em microsserviços).

Em sistemas que já estão em produção e que, consequentemente, possuem uma base robusta de clientes, alterações estruturais desse tipo podem acarretar riscos para toda a aplicação.

Isto porque, a depender de como os componentes do software foram estruturados, implementar um fluxo alternativo com a garantia de que novos bugs não serão introduzidos na aplicação irá demandar mais tempo e esforço. Isso, para o empregador, irá representar mais custo, seja através de horas extras ou com o recrutamento de mais desenvolvedores para apoiar a criação do projeto.

Por isso, pode-se supor que além de entregar um programa, também é importante que este seja de fácil manutenção. Por conta disso, diferentes princípios de arquitetura, padrões de design e até conceitos mais simples foram criados ao longo do tempo para contribuir com a redução do esforço inerente ao processo de desenvolvimento.

Um conceito simples famoso, por exemplo, é o *Don't Repeat Yourself* (DRY), que consiste em desenvolver de forma a evitar que o código seja duplicado, porém mantendo sua flexibilidade [12]. Esta prática, por consequência, garante a criação de componentes que possam ser reutilizados de forma que o desenvolvedor não se repita.

Além de conceitos simples, vale mencionar também práticas mais complexas que, embora escapem do tema deste trabalho, não estão mutualmente exclusivas a ele, como, por exemplo, metodologias de design como o *Domain-Driven Design* (DDD), de desenvolvimento como o *Test Driven Development* (TDD) e práticas específicas a certos paradigmas da programação, como os Padrões de Projetos.

Por outro lado, a adoção de algumas práticas sem moderação pode ocasionar em mais esforço para sua implementação e gerar complexidade de código adicional. Sendo assim apesar de ajudar na manutenção, pode atrapalhar a legibilidade e requisitar um conhecimento técnico mais avançado para compreender estruturas de código que, embora estejam preparadas para tal evento, talvez nunca sofram as drásticas alterações que motivaram a dita complexidade em primeiro lugar.

Com isso, também há princípios populares advertem o desenvolvedor a não exagerar com alguns cuidados. Por exemplo o *Keep It Simple, Stupid* (KISS) [13], que sugere que a implementação minimalista tende a ser mais bem sucedida que soluções complicadas, ou o *You Ain't Gonna Need It* (YAGNI), que aconselha não tentar prever supostas futuras necessidades sem a garantia que as mesmas existam em primeiro lugar [14].

Por isso, este projeto tem como objetivo aplicar e explicitar, na prática, implementações de *Clean Architecture* e *SOLID* em uma aplicação web. Ao longo deste trabalho serão expostos os desafios e benefícios de aplicar tais técnicas no desenvolvimento de uma loja virtual de jogos eletrônicos, como a *Steam* ou a *Epic Games*.

A loja será desenvolvida usando as seguintes tecnologias:

- *Back-end*: *Node.js* com *Typescript*
- *Front-end*: *React.js*
- Banco de dados: *PostgreSQL*

O segundo capítulo é dedicado a apresentar os conceitos de *SOLID* e *Clean Architecture*, além de exemplificar como essas técnicas podem ajudar a mitigar os riscos apontados no início dessa introdução.

O terceiro capítulo terá como foco descrever com mais profundidade as tecnologias que serão implementadas no desenvolvimento do software e o que motivou a escolha das mesmas.

No quarto capítulo, será introduzida a proposta da loja de jogos eletrônicos, assim como seus requisitos, regras de negócio, diagramas e toda a documentação produzida para o propósito desse trabalho.

O Capítulo 4 relacionará o conteúdo dos capítulos anteriores, expondo, no código-fonte, todas as práticas que foram possíveis de implementar dentro do escopo da loja virtual com as tecnologias apresentadas. Além disso, também será simulado alguns dos problemas exemplificados no início desse capítulo e como a adoção das práticas do tema deste trabalho ajudaram a superar os ditos problemas.

Por fim, o quinto capítulo trará as conclusões obtidas ao longo do processo de desenvolvimento do trabalho assim como descreverá possíveis trabalhos futuros de acordo com a conclusão obtida.

2 CONCEITOS

Este capítulo tem como objetivo apresentar os conceitos de *Clean Architecture* e *SOLID*.

2.1 CLEAN ARCHITECTURE

Antes de descrevermos do que se trata a arquitetura limpa, é importante descrever o próprio conceito de arquitetura de software.

2.1.1 A ARQUITETURA DE SOFTWARE

A Arquitetura de software de um sistema se refere às decisões de design relacionadas a estrutura e comportamento geral do sistema. A arquitetura ajuda stakeholders a entenderem e analisarem como o sistema irá alcançar características essenciais como manuseabilidade, disponibilidade e segurança [2].

Ao desenvolver continuamente sem se preocupar com essas decisões, as características acima se tornam mais difíceis de serem alcançadas, ocasionando problemas no longo prazo que decorrem justamente da ausência dessas características. Um software com pouca manuseabilidade irá requirir mais esforço e mais mão de obra para se dar manutenção, enquanto um sistema com pouca segurança terá um comportamento inconsistente e alta incidência de bugs. A combinação dos dois

fatores anteriores acarretam em custos desnecessários e tempo perdido testando e retestando fluxos já desenvolvidos previamente.

Dito isso, de acordo com Robert C. Martin, o objetivo de uma boa arquitetura de software "...é minimizar os recursos humanos necessários para construir e manter um determinado sistema" [1, p.5].

2.1.2 A ARQUITETURA LIMPA

Martin menciona que toda arquitetura limpa tem a Separação de Preocupações (*Separation of Concerns*) [1, p.202]. Esta consiste em segregar todo o código em módulos de acordo com suas preocupações, ou seja, quais classes ou outros detalhes da aplicação este teria acesso. O conceito da arquitetura limpa reúne esses módulos em diferentes níveis de acordo com sua proximidade das regras de negócio de uma empresa, sendo o software de nível mais baixo relacionado apenas a ferramentas externas, e o mais alto, relacionado diretamente às regras de negócio.

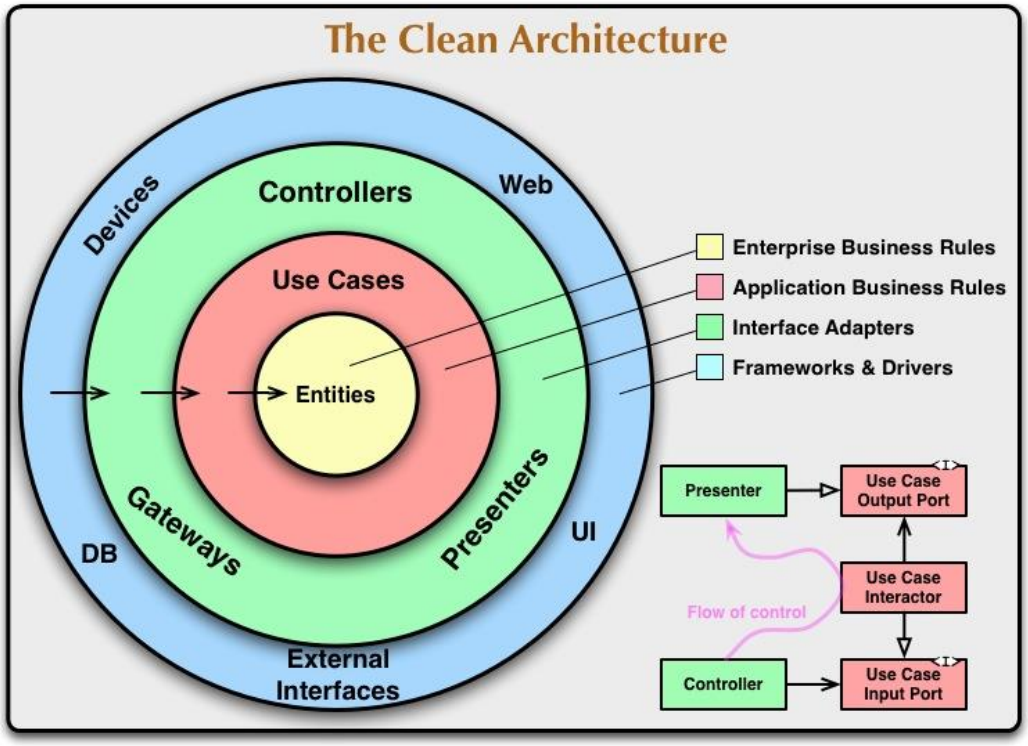

Figura 1: A arquitetura limpa [4]

O importante sobre essa divisão em níveis é garantir que mudanças nas camadas inferiores não afetem as camadas superiores. Enquanto uma mudança a nível de regra de negócio pode interferir na forma que um dado é manipulado ou é exibido, o contrário não deveria ser verdade. Esse seria o conceito de Regra da Dependência, que consiste em garantir que as dependências do código partam apenas do nível inferior para o superior.

Esses níveis podem ser representados como círculos concêntricos, de forma que as camadas de nível mais alto do código estejam mais ao centro do círculo, tal como a Figura 1.

A seguir, será detalhado do que cada nível se trata nesse modelo, e, a fim de elucidar cada camada, usaremos como exemplo um sistema de gerenciamento de usuários.

Regras de Negócio de Empresa (*Enterprise Business Rules*), ou as Entidades, são objetos com suas próprias funções ou estruturas de dados que possam

ser utilizadas por várias aplicações diferentes numa mesma empresa. No contexto desse trabalho e na maioria das aplicações web, as entidades serão as tabelas do banco de dados, devido a sua grande tendência de reutilização em diferentes partes de um mesmo sistema. No exemplo proposto, a entidade seria uma tabela de usuários com os atributos do mesmo, como o nome, endereço, login e senha.

Regras de Negócio da Aplicação (*Application Business Rules*), ou os Casos de Uso, são todos os fluxos específicos de uma aplicação. Essas regras de negócio utilizam-se manipulam as Entidades a fim de processar o que lhes for pedido. No exemplo proposto, os casos de uso seriam os algoritmos de cadastro, autenticação, consulta, atualização, remoção e outras tarefas referentes à tabela previamente apresentada.

Os Adaptadores de Interface (*Interface Adapters*) têm como propósito converter os dados entre os níveis vizinhos de uma forma que seja conveniente para todos os envolvidos. Um exemplo de adaptador no sistema pode ser uma classe responsável por tratar as requisições recebidas pelos Frameworks a fim de executar os casos de uso para gerenciar os usuários.

Por fim, os *Frameworks e Drivers* são geralmente, no contexto de desenvolvimento web, as ferramentas usadas para acesso a banco de dados e web. Por serem ferramentas já prontas, geralmente só é desenvolvido nessa camada a comunicação com os níveis mais internos.

Em resumo, um sistema que separa suas preocupações e que segue a Regra da Dependência tendem ter uma arquitetura limpa, com módulos de lógica de negócios testáveis, independentes de *frameworks*, UI, banco de dados ou qualquer agência externa.

2.2 SOLID

Os princípios SOLID, em resumo, se tratam de regras de organização de funções e estruturas de dados em agrupamentos. O principal objetivo de sua aplicação está em criar software que seja de fácil entendimento, manutenção e reaproveitamento.

Também proposto por Robert C. Martin [3], SOLID é um acrônimo para siglas que representam princípios a serem seguidos no desenvolvimento de um bom software orientado a objetos. Para exemplificar cada um desses princípios, consideremos um sistema de pagamentos.

2.2.1 *SINGLE RESPONSABILITY PRINCIPLE*

De acordo com o SRP, ou o Princípio da Responsabilidade Única, uma classe deveria ter uma, e somente uma razão para ser alterada [3]. Esse princípio é relevante, pois ao dividir as responsabilidades corretamente em cada módulo, evita-se consequências não planejadas durante o processo de manutenção do software.

Exemplificando essa situação, ao desenvolver uma classe que é responsável tanto por gerenciar usuários quando efetuar pagamentos, estamos quebrando esse princípio. Imagine que é realizada uma validação do usuário destinatário antes de se executar um pagamento. Se houver alguma mudança nessa rotina de validação, é possível que isso afete a rotina de pagamentos de formas imprevistas. A forma que seguiria o SRP seria desenvolver uma classe que fosse responsável pela gerência dos usuários e outra que fosse responsável pelos pagamentos.

2.2.2 *OPEN-CLOSED PRINCIPLE*

O Princípio Aberto-Fechado consiste em garantir que o comportamento de uma classe deveria estar aberto para a extensão, mas fechado para modificações [3]. A importância desse princípio se deve ao fato de que ao modificar uma classe, essa modificação provavelmente vai influenciar algum ponto não previsto do sistema que utilize esse mesmo componente.

No sistema exemplificado, ao acrescentar suporte a pessoas jurídicas ao algoritmo de validação de usuários, se simplesmente acrescentarmos ao código existente uma condição para verificar o tipo da pessoa (Física ou Jurídica) a fim de escolher o método de validação, estaríamos quebrando o OCP. O correto seria termos uma classe abstrata Pessoa e estender esse comportamento através de duas classes filhas PessoaFisica e PessoaJuridica, cada uma com seu método de validação, conforme a Figura 2.

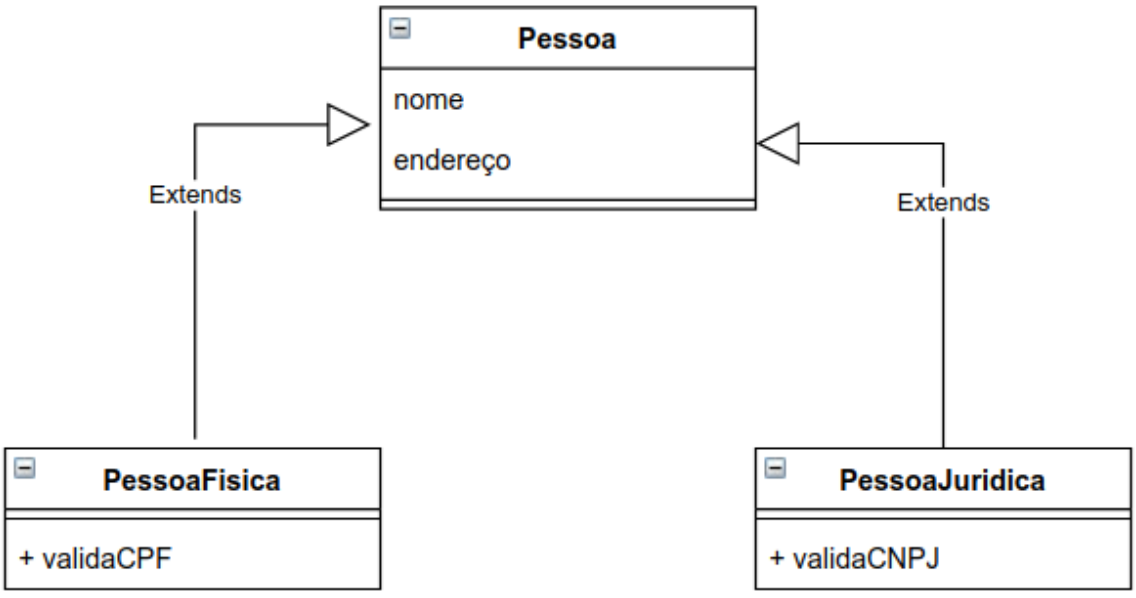

Figura 2: Relação entre PessoaFisica e PessoaJuridica

2.2.3 *LISKOV SUBSTITUION PRINCIPLE*

Segundo o Princípio da Substituição de Liskov, as classes devem ser capazes de serem substituídas por suas classes derivadas sem afetar a execução do programa (e vice-versa) [3]. Este princípio, aliado ao OCP, garante uma maior facilidade de manutenção do código.

Reutilizando os exemplos anteriores, o desenvolvedor aplicará o LSP se o mesmo passar o método de validação de uma classe Pessoa ao tentar validar a PessoaFisica ou PessoaJuridica antes de realizar um pagamento.

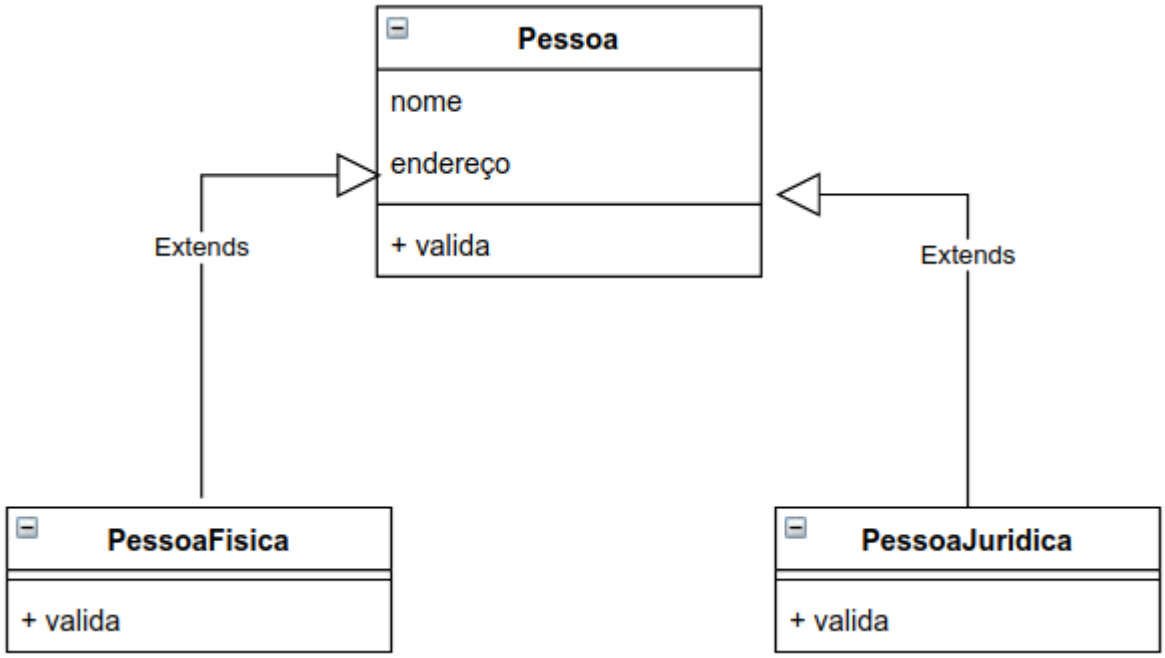

Figura 3: Renomeando métodos para adequação ao LSP

Conforme descrito na Figura 3, ao garantir que a classe Pessoa e suas classes herdeiras tenham o método de mesmo nome, podemos garantir um nível de intercambiabilidade entre elas. Ou seja, em fluxos onde o tipo de pessoa não é uma informação relevante, podemos usar uma classe genérica (neste caso, Pessoa) para reaproveitar melhor o código.

2.2.4 *INTERFACE SEGREGATION PRINCIPLE*

No Princípio de Segregação de Interfaces define que as interfaces devem ser criadas de forma granular e específica para seus clientes. Isso quer dizer que não é interessante um cliente (ou classe) depender de uma interface que possui métodos que o mesmo não tenha conhecimento, ou, de forma simplificada, que interfaces não deveriam ter métodos que não serão usados por suas implementações [3].

Por exemplo, havendo uma interface IPagamento e duas classes PagamentoBoleto e PagamentoTed, o ISP estaria sendo violado se houver um método getLinhaDigitavel na IPagamento, visto que pagamentos via TED não usam linhas digitáveis. O ideal seria que essa interface fosse mais bem subdividida (ou granularizada) a fim de que as classes usassem todos os seus métodos, como na Figura 4.

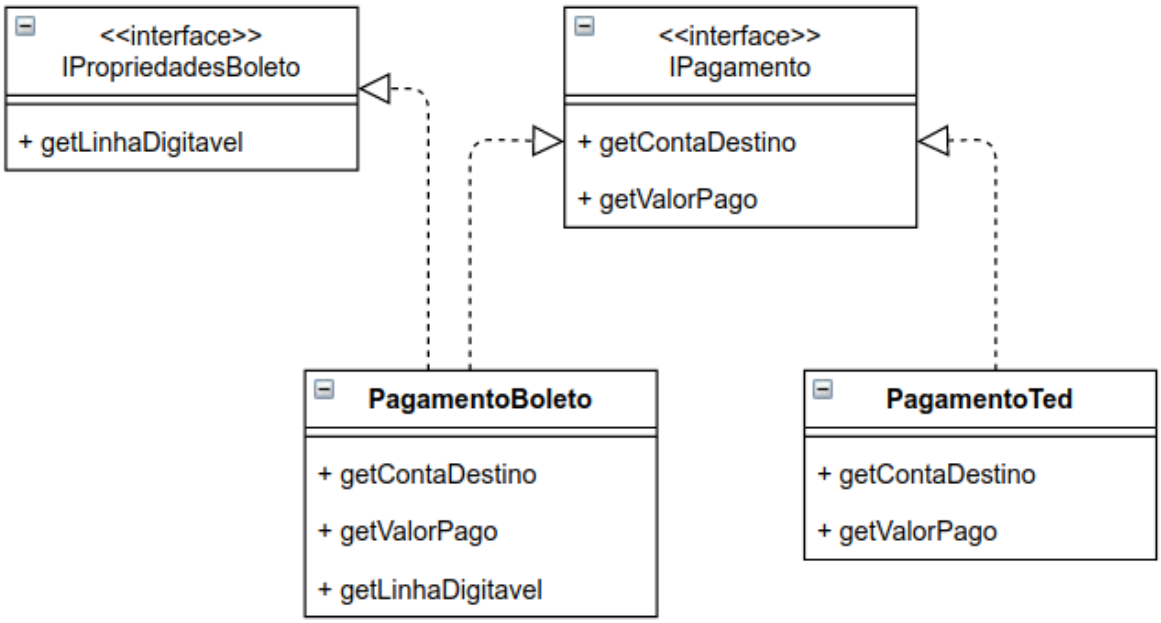

Figura 4: Exemplo do ISP

2.2.5 *DEPENDENCY INVERSION PRINCIPLE*

Por fim, o Princípio da Inversão de Dependências indica que módulos de níveis superiores deveriam depender apenas de abstrações de módulos de níveis inferiores, e não de implementações [3]. Este princípio está fortemente ligado ao modelo de Arquitetura Limpa descrito na Seção 2.1.2, principalmente na relação entre a camada de Casos de Uso e a camada de Adaptadores de Interfaces.

Suponha-se que o exemplificado sistema de pagamentos consuma algum serviço externo para realizar alguma consulta ou validação. A estratégia de implementação da chamada a esse serviço externo, seguindo o DIP, seria criar uma interface que estabelecesse o contrato com o serviço externo e usá-la no sistema de pagamentos toda vez que ele fosse necessário.

3 TECNOLOGIAS

Neste capítulo serão abordadas as tecnologias usadas no desenvolvimento da aplicação *web*. A linguagem que permeará tanto o *frontend* quanto o *backend* será o *JavaScript*, porém, como as práticas abordadas neste trabalho são mais direcionadas ao *backend*, não será descrito com detalhes as tecnologias do *frontend*.

Além disso, como um dos temas principais desse trabalho está em como o código se comporta em relação a suas dependências, será abordado também os *frameworks web* e de banco de dados, assim como uma ferramenta ajudar a deixar a arquitetura desse código mais limpa.

3.1 JAVASCRIPT E TYPESCRIPT

O *JavaScript*, também conhecido como *ECMAScript* ou JS, é uma linguagem leve e interpretada que pode ser executada tanto *client-side*, como nos navegadores, quanto *server-side*, como em servidores remotos.

Na época de sua padronização, era esperado que fosse apenas uma simples linguagem de scripts para navegadores, apenas para usos pontuais em páginas web [8]. Porém, com o tempo, a linguagem se tornou mais popular ao longo de que desenvolvedores começaram a usá-la com mais frequência para aprimorar a interacionabilidade de páginas web e hoje a mesma extrapola o ambiente dos navegadores e roda também nos servidores através do Node.js (Seção 3.2).

Porém, seu design original não previa o uso em aplicações muito complexas, e mesmo com as atualizações mais recentes do *ECMAScript*, a linguagem pode

acabar sendo uma opção insegura para algumas aplicações empresariais por, dentre alguns motivos, sua fraca tipagem.

Por essa desvantagem característica do *JavaScript*, o *TypeScript* surge como uma proposta de ser uma camada sobre o *JavaScript* que permita checagem estática de tipos. Desta forma, a linguagem garante, durante o desenvolvimento, alguns recursos que permitam uma maior previsibilidade em relação ao resultado do código desenvolvido sem alterar o comportamento de execução do *JavaScript*.

3.2 *NODE.JS*

O *Node.js* é um ambiente de execução *JavaScript* assíncrono e orientado a eventos. Apesar de ser *single threaded*, o *Node.js* possui o *event loop*, que permite o mesmo a executar operações E/S de forma não bloqueante ao delegar as mesmas para o *kernel* do sistema sempre que possível. Ou seja, o *event loop* garante que não haja a necessidade de que o código JavaScript espere o término da execução de algum processo não-*JavaScript* para executar a próxima instrução [5].

Devido a esse diferencial, o *Node.js* pode lidar com um número elevado de requisições simultâneas em um único servidor sem a necessidade de engessar o projeto com a gerência de *threads* simultâneas, o que simplifica o trabalho do desenvolvedor.

Uma outra vantagem em relação a outros ambientes é que, por aplicações *Node.js* serem escritas em *JavaScript*, parte do aprendizado necessário para desenvolver para o *frontend* acaba sendo reutilizado no *backend* sem a necessidade de aprender uma nova linguagem de programação para o uso no servidor.

3.3 BIBLIOTECAS E FRAMEWORKS

Nesta seção serão descritos os *frameworks* ou bibliotecas mais relevantes para o *backend* desta aplicação. No caso, apenas aqueles que são cruciais para o funcionamento da mesma.

3.3.1 TYPEORM E POSTGRESQL

O Sistema de Gerenciamento de Banco de Dados escolhido para este trabalho foi o PostgreSQL. Dentre as razões que motivam essa escolha [9]:

- Seu uso é gratuito
- É popular
- *Open Source* com uma comunidade ativa
- É um projeto sólido com mais de 30 anos de desenvolvimento
- Bastante compatibilidade com ORMs das tecnologias apresentadas

Para manipularmos o acesso ao banco de dados PostgreSQL, iremos um utilizar uma ferramenta Object-Relational Mapping, ou ORM. O que motiva esse uso é a facilidade que as ferramentas ORMs pode trazer ao processo de desenvolvimento ao abstrair completamente a camada de comunicação com o Banco de Dados.

Isto é, o desenvolvedor não precisa conhecer muitos detalhes de implementação ou sintaxe de algum Sistema de Banco de Dados específico, além de não precisar se preocupar com a sanitização das queries ou com outros aspectos relacionados às chamadas ao banco de dados.

O ORM utilizado nesse trabalho será o *TypeORM*, um framework voltado principalmente para o *TypeScript*. A preferência por esse ORM se dá por vários motivos. Primeiramente, a compatibilidade com os principais bancos de dados relacionais, com o *MongoDB* (em estado experimental) e com as principais plataformas onde o *JavaScript* é suportado (*Node.js*, Navegador, *React Native*, entre outros) [10].

Mas o motivo principal para a escolha desse ORM está na sua adoção pelo padrão *Data Mapper*. Este padrão de ORM está bastante relacionado com as ideias de facilidade de manutenção já abordadas no Capítulo 2, visto que o uso do mesmo promove a separação de preocupações e, por consequência, baixo acoplamento e maior qualidade de código [10][11].

3.3.2 EXPRESS E REACT

O *Express Js* é um dos *frameworks web* mais populares para o ambiente *Node.js* [6]. Sua utilização neste trabalho será como um intermediador entre as requisições enviadas pela página *web* e as informações necessárias para a nossa aplicação.

Note que para o desenvolvimento desse trabalho ou de qualquer aplicação web, usar o *Express* ou qualquer outro *framework web* não é estritamente necessário. Porém, essas ferramentas agilizam bastante o processo de desenvolvimento, logo o uso destas é recomendado.

O *React* é um framework web voltado para o *front-end*, ou seja, a *User Interface* disponibilizada através da página *web*. Como o conceito de dependência predomina neste trabalho, é importante apresentá-lo como o único responsável pela UI e por toda UI.

Desta forma, estamos garantindo que nenhum código responsável por lógicas de negócio complexas também esteja responsável pela disposição dos elementos na página *web*, o que acaba sendo uma decisão que respeita os princípios do *SOLID* e da *Arquitetura Limpa* ao separar essas preocupações.

Novamente, para o desenvolvimento de uma boa UI, não é estritamente necessário o uso de *React*, mas devido a facilidade de componentização de objetos típicos de interfaces *web* (botões e formulários, por exemplo) e sua popularidade [6], o mesmo foi escolhido para este trabalho.

3.3.3 TSYRINGE

Por fim, o *Tsyringe* será a ferramenta que utilizaremos para injetar as dependências que serão explicitadas posteriormente nesse trabalho.

Desenvolvida pela *Microsoft* [7], o *Tsyringe* permite o uso de anotações como *@Injectable* e *@Inject*, que, através da Inversão de Controle, garantem uma manuseabilidade maior com uma sintaxe elegante para o *Typescript*.

4 SOBRE A APLICAÇÃO

Este capítulo é destinado a exibir os detalhes acerca da loja virtual, isto é, os requisitos, diagramas UML e recursos disponíveis na aplicação web.

4.1 OBJETIVO DA APLICAÇÃO

A GameStore, a aplicação desenvolvida durante este trabalho, tem como sua principal finalidade representar uma loja de jogos, tal como a *Steam* ou a *Epic Games.* Para garantir o funcionamento da loja, é necessário que seja possível cadastrar usuários e que estes possam comprar produtos na GameStore.

Para evitar de implementar APIs que executem transferência bancárias reais, foram desenvolvidas funcionalidades de geração e reivindicação de cartões pré-pagos a fim de garantir que os usuários consigam comprar itens na loja a partir de um dinheiro virtual.

4.2 REQUISITOS DA APLICAÇÃO

Para o MVP (Mínimo Produto Viável) deste projeto, focaremos nas funcionalidades mais básicas que se pode esperar de uma plataforma de jogos, isto é, interagir com os jogos. Sendo assim, a loja deveria permitir:

- Exibir jogos registrados na plataforma

Além disso, é necessário manter um registro dos usuários que acessam essa plataforma e dos jogos comprados pelos mesmos, então a loja também deve permitir que seus usuários possam:

- Se cadastrar
- Se autenticar
- Redefinir os parâmetros de seu perfil
- Comprar jogos registrados na plataforma
- Listar os jogos adquiridos
- Resgatar cartões pré-pago
- Consultar histórico de compras

Alguns fluxos estarão disponíveis apenas para uso interno, tais como:

- Registro de novos jogos
- Cadastro de cartões pré-pago

4.3 DETALHES DA APLICAÇÃO

4.3.1 MODELOS E DIAGRAMAS

Será apresentado nessa seção os detalhes referentes a aplicação que representam o funcionamento dos recursos (ou *features*) levantados na Seção 4.2.

Primeiramente, o modelo lógico de dados, ou seja, a forma como as entidades e relacionamentos do banco de dados da aplicação foram construídos, está no anexo A, na Figura 11. Em segundo lugar, os Diagramas de Fluxos dos Casos de Uso da aplicação estão representados no anexo B.

4.3.2 JORNADA DO USUÁRIO

A Jornada do usuário da aplicação, isto é, a ordem de todos os fluxos da aplicação de forma que uma pessoa consiga se cadastrar em usar 100% do sistema, é dada pela seguinte ordem dos fluxos:

Primeiramente, o usuário se cadastra e autentica através dos fluxos descritos, respectivamente, na Figura 13 e Figura 12. Após isso, o mesmo pode, opcionalmente, redefinir parâmetros de seu perfil, como alterar seu nome ou apelido no sistema, através do fluxo descrito na Figura 15. Na tela de redefinição, assim como em outros pontos da aplicação, é usado o fluxo descrito na Figura 14, responsável pela obtenção dos dados do usuário.

O usuário já cadastrado pode verificar os jogos listados na loja através do fluxo na Figura 19 e acessar os detalhes do mesmo através caso de uso da Figura 18. Como esse usuário recém-criado não tem créditos na loja, ele precisa reivindicar cartões pré-pagos através do caso de uso da Figura 22 para abastecer sua carteira virtual e, enfim, ser capaz de comprar os jogos da loja. Através de uma funcionalidade de carrinho construída exclusivamente para o *front end*, o usuário, agora com créditos, pode executar o fluxo descrito na Figura 23 e comprar os jogos da plataforma.

Os itens comprados na loja podem ser listados pelo usuário em sua biblioteca a partir do fluxo da Figura 25. As compras feitas pelo mesmo podem ser listadas em seu histórico a partir do caso de uso da Figura 24.

4.3.3 CASOS DE USO RESTRITOS

Por fim, temos os casos de uso que são restritos a administradores da aplicação. Nenhum dos fluxos a seguir pode ser chamado pelo *front-end* da aplicação, todos eles são acessíveis apenas por APIs internas.

Primeiramente, é imprescindível para o funcionamento da loja que a mesma tenha os produtos a serem vendidos, logo temos um fluxo de inserção e atualização de jogos descritos na Figura 16 e Figura 17.

Para garantir que os usuários possam comprar produtos na loja, temos o fluxo de resgate de créditos (Figura 20) e, alternativamente, o fluxo de reivindicação de jogos (Figura 21).

5 USANDO OS CONCEITOS EM UMA APLICAÇÃO WEB

O propósito desse capítulo é demonstrar como os conceitos explicitados no Capítulo 2 podem ser utilizados na construção de uma aplicação web.

5.1 DESENVOLVENDO A ARQUITETURA LIMPA

Na Seção 2.1.2 foi explicado a Regra da Dependência e o diagrama da Arquitetura Limpa (Figura 1). Esses princípios foram mantidos na aplicação desenvolvida para este trabalho e, para exemplificar o uso desses conceitos, tomamos como exemplo o fluxo de inserção de novos jogos na plataforma.

5.1.1 A IMPLEMENTAÇÃO

Primeiramente, temos as Regras de Negócio de Empresa, ou, como implementado na aplicação, as Entidades. No contexto delineado, a Entidade abordada é a classe Game. Assim como descrito na Seção 2.1.2, a Entidade Game (Código 1), como um componente de alto nível, não deve depender de nenhum outro ponto da aplicação web e é, basicamente, uma estrutura de dados que pode ser utilizada por mais de uma aplicação da empresa.

```typescript
@Entity('games')
export default class Game {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column()
  name: string;

  @Column()
  price: number;

  @Column()
  publisher: string;

  @Column()
  release_date: Date;

  @CreateDateColumn()
  created_at: Date;

  @UpdateDateColumn()
  updated_at: Date;
}
```

Código 1: A Entidade Game

Já a classe que será o Caso de Uso, ou a Regra de Negócio da Aplicação, é a CreateGameService (Código 2). Nota-se que essa classe possui como dependência a entidade Game, assim como previsto na Regra da Dependência, mas não deve depender de nenhum módulo de nível inferior.

```
@injectable()
export default class CreateGameService {
  constructor(
    @inject('GamesRepository')
    private gamesRepository: IGamesRepository,
  ) {}

  public async execute(data: ICreateGameDTO): Promise<Game> {
    //validacoes
    return await this.gamesRepository.create(/* dados do jogo */);
  }
}
```

Código 2: O Caso de Uso

A princípio, é plausível assumir que esta classe estaria violando a Regra da Dependência ao importar o módulo IGamesRepository (Código 3), visto que Repositories são módulos de nível inferior aos Casos de Uso. Porém, ao inspencionar o conteúdo deste módulo, verifica-se que o mesmo se trata de uma interface. Isso é a inversão de dependência, que será discutida em detalhes em outra seção.

```
export default interface IGamesRepository {
  index(): Promise<Game[]>;
  findById(id: string): Promise<Game | undefined>;
  findByNameAndPublisherAndReleaseDate(
    data: ICreateGameDTO,
  ): Promise<Game | undefined>;
  create(data: ICreateGameDTO): Promise<Game>;
  save(game: Game): Promise<Game>;
}
```

Código 3: A interface IGamesRepository

Por fim, temos uma classe que implementa a interface acima. O módulo GamesRepository (Código 4) será uma das classes responsáveis por se conectar com o TypeORM, um dos Frameworks ou Drivers de nossa aplicação. Por isso, essa classe se encaixa no arquétipo de Adaptadores de Interface.

```
export default class GamesRepository implements IGamesRepository {
  private ormRepository: Repository<Game>;

  constructor() {
    this.ormRepository = getRepository(Game);
  }

  public async findByNameAndPublisherAndReleaseDate({
    name,
    publisher,
    release_date,
  }: ICreateGameDTO): Promise<Game | undefined> {
    log.debug('Games :: findByNameAndPublisherAndReleaseDate');
    return this.ormRepository.findOne(/* Parametros de busca*/);
  }

  public async create(gameData: ICreateGameDTO): Promise<Game> {
    log.debug('Games :: create');
    const game = this.ormRepository.create(gameData);
    await this.ormRepository.save(game);
    return game;
  }
}
```

Código 4: O Adaptador de Interface do Banco de Dados

Até aqui, todos os níveis apresentados na Figura 1 já foram representados através das classes mencionadas nesta seção, do nível mais alto da aplicação até a dependência da camada de banco de dados. Para completar este fluxo, resta apenas exibir a camada web.

A classe GamesController (Código 5) tem uma dependência explícita do módulo CreateGameService. Além disso, também depende de outros casos de uso da aplicação, como a atualização de dados de jogos (UpdateGameService) e a listagem ou exibição de detalhes dos mesmos (IndexGameService e ShowGameService).

Assim como a classe GamesRepository, a GamesController é um tipo de Adaptador de Interface que se comunica diretamente com o *Express JS*, nosso *framework* que lida com as requisições *HTTP*.

```
export default class GamesController {
  public async index(request: Request, response: Response):
Promise<Response> {
    const indexGame = container.resolve(IndexGameService);
    const games = await indexGame.execute();

    return response.json(games);
  }

  public async create(request: Request, response: Response):
Promise<Response> {
    const createGame = container.resolve(CreateGameService);
    const game = await createGame.execute(/* dados da requisicao*/);

    return response.json(game);
  }

  public async show(request: Request, response: Response): Promise<Response>
{
    const showGame = container.resolve(ShowGameService);
    const game = await showGame.execute((/* dados da requisicao*/);

    return response.json(game);
  }

  public async update(request: Request, response: Response):
Promise<Response> {
    const updateGame = container.resolve(UpdateGameService);

    const game = await updateGame.execute((/* dados da requisicao*/);
    return response.json(game);
  }
}
```

Código 5: Adaptador de Interfaces da Web

Na Figura 5, temos a representação gráfica de como os módulos citados se situam no modelo clássico de arquitetura limpa e como eles dependem um dos outros. As setas representam as dependências, e elas partem das classes de nível inferior até as classes de nível superior. Nota-se que o contrário não ocorre.

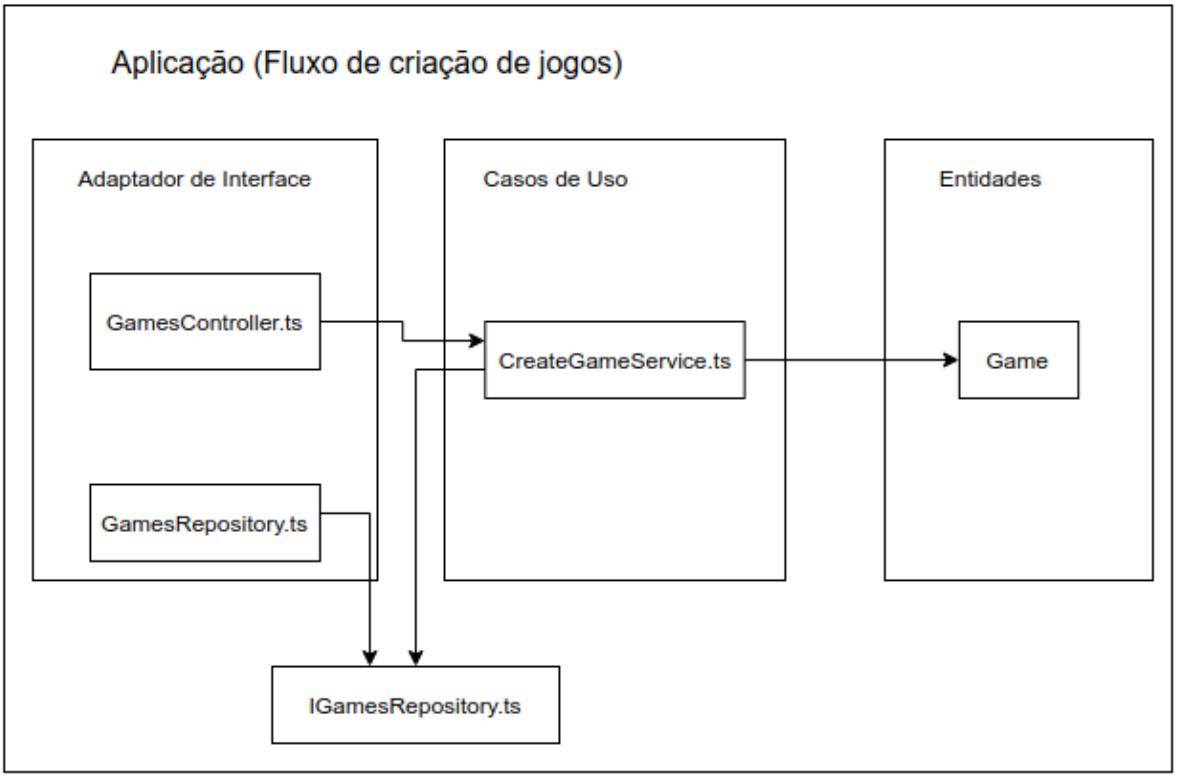

Figura 5: Aplicação Limpa

5.1.2 OS BENEFÍCIOS

O benefício de adotar esse método de desenvolvimento se torna óbvio ao necessitar reutilizar o caso de uso em algum outro fluxo da aplicação. Neste caso, supondo que haja a necessidade de cadastrar grandes volumes de jogos na nossa plataforma, seria mais conveniente passar toda essa informação através de um arquivo do que inseri-la manualmente através de requisições Web. Por conta disso, foi criado uma nova classe, GamesCreateBatch (Código 6), que processa uma planilha em um arquivo csv e invoca o serviço de criação de games para cada linha.

```
export default class GamesCreateBatch implements IBatchOperation {
  public async exec(): Promise<void> {
    //valida o nome do arquivo passado via linha de comando
    //cria lista de linhas

    const createGame = container.resolve(CreateGameService);

    await Promise.all(lines //percorre cada linha da lista criada
      .map(async (line: string): Promise<Game | boolean> => {
        try {
          //validacao da linha

          return await createGame.execute(/* conteudo da linha */);
        } catch (error) {
          return log.error(error);
        }
      }));
  }
}
```

Código 6: Caso de Uso reaproveitado para processamento de arquivos

Como os níveis desse fluxo foram muito bem segregados ao longo dos módulos, não foi necessário alterar nenhuma linha de código das classes responsáveis por conhecer as regras de negócios da aplicação e das responsáveis por interagir com o banco de dados.

Outro ponto é que se em algum momento for necessário trocar o Express JS, somente a classe GamesController (Código 5) deverá sofrer uma manutenção, o restante, pelo menos referente a esse fluxo, continuará intacto.

Outro ganho da adoção desse método está na facilidade em testar os componentes dessa aplicação. Para isso, foi criado a classe FakeGamesRepository (Código 7), cujo propósito está em simular o acesso ao banco de dados. Como essa classe é uma implementação da interface IGamesRepository, não haverá a necessidade de realizar nenhuma alteração no Caso de Uso para se realizar dos testes ou, caso seja necessário, para trocar o *TypeORM* por algum outro framework de acesso a banco de dados.

```
export default class FakeGamesRepository implements IGamesRepository {
  private games: Game[] = [];

  public async findByNameAndPublisherAndReleaseDate(data: ICreateGameDTO):
Promise<Game | undefined> {
    return this.games.find(game =>
      game.name === data.name &&
      game.publisher === data.publisher
      game.release_date === data.release_date
    );
  }

  public async create(gameData: ICreateGameDTO): Promise<Game> {
    const game = new Game();
    Object.assign(game, { id: uuidv4() }, gameData);
    this.games.push(game);
    return game;
  }
```

Código 7: Outra implementação do IGamesRepository

5.2 DESENVOLVENDO UMA APLICAÇÃO SOLID

No fluxo descrito na seção anterior já é possível encontrar alguns conceitos SOLID sendo implementados na aplicação web. Porém, será necessário ver outros fluxos para obter exemplos de implementação que sigam todos os princípios.

5.2.1 IMPLEMENTAÇÂO DO SRP

O Princípio da Responsabilidade Única (SRP) está sendo seguido na classe ShowGameService (Código 8). Isso ocorre, pois, a classe tem apenas um propósito, que é a exibição dos detalhes do jogo. Sendo assim, a mesma tem um

escopo mais restrito em razões para se necessitar de alteração, por exemplo, adicionar validações, passar a buscar os jogos pelo nome em vez do id, entre outros.

```
@injectable()
export default class ShowGameService {
  constructor(
    @inject('GamesRepository')
    private gamesRepository: IGamesRepository,
  ) {}
  public async execute(data: IRequest): Promise<Game> {
    //validacao e retorno do jogo
  }
}
```

Código 8: Implementação correta do SRP

Uma forma fácil de imaginar a classe acima ferindo este princípio seria se a mesma além de exibir jogos, também criasse ou atualizasse os mesmos. Com isso, a manutenção dessa classe pode ser mais difícil de se realizar do que a da primeira:

```
@injectable()
export default class GameService {
  constructor(
    @inject('GamesRepository')
    private gamesRepository: IGamesRepository,
  ) {}
  public async showGame(data: IShowGameDTO): Promise<Game> {
    //validacoes e processos para exibir detalhes de um jogo
  }
  public async createGame(data: ICreateGameDTO): Promise<Game> {
    //validacoes e processos para criar um jogo
  }
  public async updateGame(data : IUpdateGameDTO): Promise<Game> {
    //validacoes e processos para atualizar um jogo
  }
}
```

Código 9: Implementação que viola o SRP

Ao absorver muitas responsabilidades, a classe GameService (Código 9) passa a englobar diversos comportamentos e, com isso, passa a exigir mais esforço para se manusear.

Uma forma que respeite o SRP e implemente as funcionalidades da classe GameService pode ser descrita na Figura 6: Três classes, cada uma com suas regras de negócio e suas responsabilidades.

Figura 6: Representação do SRP

Desta forma, o leitor pode se questionar porque a classe GamesController (Código 5) não viola o SRP, visto que a mesma realiza todas essas operações. Isso não se caracteriza como uma violação pois a responsabilidade da classe é apenas comunicar as chamadas recebidas via HTTP para a aplicação. O Controller desconhece os fluxos de exibição, criação ou atualização, mas conhece as classes que contêm esses fluxos.

5.2.2 IMPLEMENTAÇÃO DO OCP

Para exemplificar o Princípio Aberto Fechado (OCP), usaremos o fluxo responsável pela reivindicação de cartões pré-pagos. Para usar esse serviço, o usuário insere o código de cartão um cartão pré-pago e resgata jogos ou dinheiro para a compra de jogos na loja.

Porém, ao cadastrar esses códigos no sistema, a validação para esses dois tipos de itens a ser reivindicados é diferente. Os cartões de dinheiro devem resgatar somente os valores de R$30, R$50 ou R$100. Para validar um jogo, somente é necessário que este exista na base de dados. Com os requisitos acima, um desenvolvedor pode pensar em criar a classe descrita no Código 10:

```
@injectable()
export default class CreateCodeService {
  constructor(
    @inject('GamestoreCodesRepository')
    private gamestoreCodesRepository: IGamestoreCodesRepository,

    @inject('GamesRepository')
    private gamesRepository: IGamesRepository,
  ) {}

  public async execute(request: IRequest): Promise<GamestoreCode |
undefined> {
    if (!request.cash && !request.game_id) {
      throw new AppError('Internal Server Error', 500);
    }

    if (request.cash) {
      //validacao de cash (R$30, R$50 ou R$100)
      //cadastro do cartao pre pago de cash
    }

    if (request.game_id) {
      //validacao de game (se o game existe)
      //cadastro do cartao pre pago de gane
    }
  }
}
```

Código 10: Implementação de cadastro de códigos

Embora atenda a demanda, essa estratégia fere o OCP. Note que, se em algum momento no futuro for necessário reivindicar um terceiro item, teremos que adicionar mais uma condição ao método principal dessa classe, o que deixará o código ainda mais acoplado. Para ilustrar isso, suponhamos que será necessário reivindicar códigos que resgatem Pacotes de Expansão para jogos já cadastrados em

nossa plataforma. Essa funcionalidade extrapola o escopo desse trabalho, mas sua implementação pode ser descrita no Código 11.

```
@injectable()
export default class CreateCodeService {
  constructor(
    @inject('GamestoreCodesRepository')
    private gamestoreCodesRepository: IGamestoreCodesRepository,
    @inject('GamesRepository')
    private gamesRepository: IGamesRepository,
    @inject('DlcsRepository')
    private dlcsRepository: IDlcsRepository,
  ) {}

  public async execute(request: IRequest): Promise<GamestoreCode |
undefined> {
    if (!request.cash && !request.game_id && !request.dlc_id) {
      throw new AppError('Internal Server Error', 500);
    }
    if (request.cash) {
      //validacao de cash (R$30, R$50 ou R$100)
      //cadastro do cartao pre pago de cash
    }
    if (request.game_id) {
      //validacao de game (se o game existe)
      //cadastro do cartao pre pago de gane
    }
    if (request.dlc_id) {
      //validacao de dlc (se a dlc existe)
      //cadastro do cartao pre pago de gane
    }
  }
}
```

Código 11: Implementação que viola o OCP

Para não ferir este princípio, basta estendermos a funcionalidade de uma classe base e realizar as validações nas classes herdeiras. A seguir temos os passos para realizar esse procedimento:

Primeiramente, definimos uma classe abstrata (Código 12) que concentrará os métodos que serão realizados pelas classes filhas desta. Como os fluxos de

cadastro de códigos para jogo e dinheiro virtual passam por processos diferentes de validação e cadastro, este serão métodos abstratos desta classe.

```
export default abstract class AbstractCodeTemplate {
    public async execute(request: IRequest): Promise<GamestoreCode> {
      await this.validateData(request);
      const code = await this.createCode(request);
      return code;
    }

    protected abstract createCode(request: IRequest):
Promise<GamestoreCode>;
    protected abstract validateData(request: IRequest): Promise<void>;
  }
```

Código 12: Classe Abstrata

Em seguida, estendemos, a partir da classe abstrata, uma classe para cada comportamento diferente. Sendo assim, uma classe para validar e registrar jogos (Código 13) e outra para dinheiro virtual (Código 14).

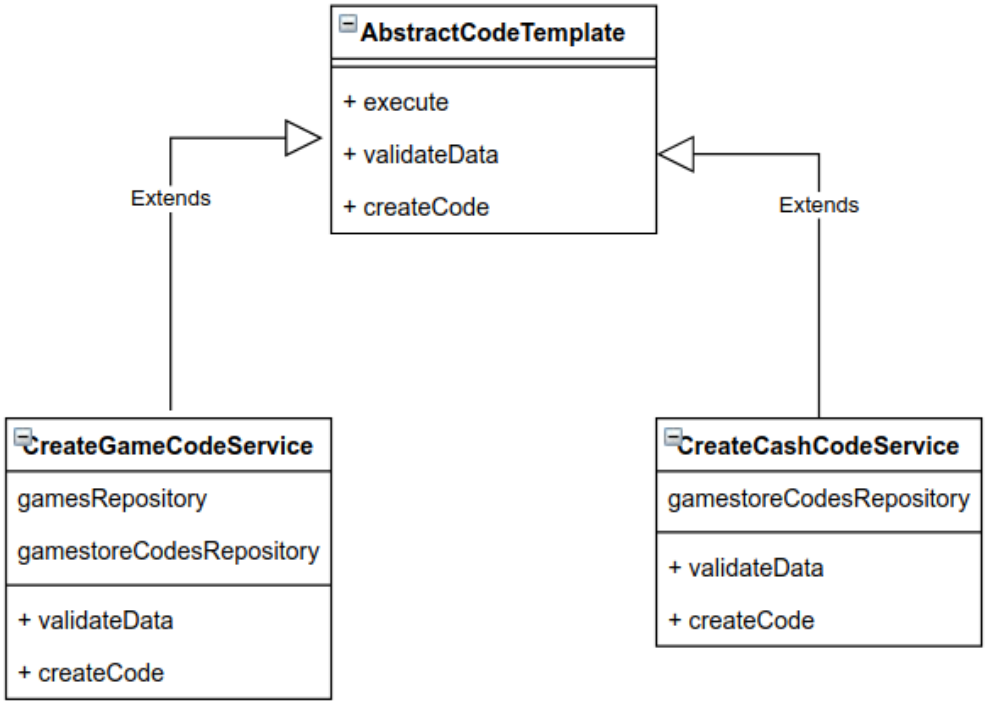

Figura 7: Representação do OCP

Na Figura 7, temos uma representação de como essas classes se relacionam. As setas partem em direção a classe abstrata, indicando a classe que estende a mesma.

```
@injectable()
export default class CreateGameCodeService extends AbstractCodeTemplate {
  constructor(
    @inject('GamesRepository')
    private gamesRepository: IGamesRepository,

    @inject('GamestoreCodesRepository')
    private gamestoreCodesRepository: IGamestoreCodesRepository,
  ) {
    super();
  }

  protected async createCode({ game_id }: IRequest): Promise<GamestoreCode>
{
    const code = await this.gamestoreCodesRepository.create(/* dados do jogo
*/)
  }

  protected async validateData({ game_id }: IRequest): Promise<void> {
    //algoritmo de validacao de jogo
  }
}
```

Código 13: Cadastro de Códigos para Jogos

```
@injectable()
export default class CreateCashCodeService extends AbstractCodeTemplate {
  constructor(
    @inject('GamestoreCodesRepository')
    private gamestoreCodesRepository: IGamestoreCodesRepository,
  ) {
    super();
  }

  protected async createCode(data: IRequest): Promise<GamestoreCode> {
    const code = await this.gamestoreCodesRepository.create(/*
      dados do dinheiro virtual
    */);
    return code;
  }

  protected async validateData(data: IRequest): Promise<void> {
    //algoritmo de validacao do dinheiro virtual
  }
}
```

Código 14: Cadastro de Códigos para Dinheiro Virtual

5.2.3 IMPLEMENTAÇÃO DO LSP

Um bom exemplo de implementação que respeita o LSP já foi apresentado na subseção anterior.

Esse princípio pode ser quebrado se, por exemplo, a interface não especificar um método requerido em algum dos casos de uso e terceirizar essa responsabilidade diretamente para as subclasses. A intercambiabilidade é perdida, e assim, a possibilidade de reuso do código também.

O benefício de se usar esse princípio está na desacoplação do código. Sem esse princípio aplicado, por exemplo, seria necessário especificar o subtipo

UsersRepository, que, como demonstrado previamente, por se tratar de um Adaptador de Interfaces, é um módulo de nível inferior aos Casos de Uso.

5.2.4 IMPLEMENTAÇÃO DO ISP

Um exemplo muito recorrente do próximo princípio, isto é, o Princípio da Segregação de Interfaces (ISP), está no funcionamento das classes *Repository* da aplicação.

```
import ICreateUserDTO from '../dtos/ICreateUserDTO';
import User from '../infra/typeorm/entities/User';

export default interface IUsersRepository {
  findById(id: string): Promise<User | undefined>;
  findByEmail(email: string): Promise<User | undefined>;
  create(data: ICreateUserDTO): Promise<User>;
  save(user: User): Promise<User>;
}
```

Código 15: Interface a ser implementada

Tomando por exemplo o Repository de Usuários, temos a interface IUsersRepository (Código 15) e suas implementações UsersRepository (Código 16) e FakeUsersRepository (Código 17).

```
export default class UsersRepository implements IUsersRepository {
  private ormRepository: Repository<User>;

  constructor() {
    this.ormRepository = getRepository(User);
  }

  public async findById(id: string): Promise<User | undefined> {
    return this.ormRepository.findOne(id);
  }

  public async findByEmail(email: string): Promise<User | undefined> {
    const user = await this.ormRepository.findOne({
      where: { email },
    });
    return user;
  }

  public async create(userData: ICreateUserDTO): Promise<User> {
    const user = this.ormRepository.create(userData);
    await this.ormRepository.save(user);
    return user;
  }

  public async save(user: User): Promise<User> {
    return this.ormRepository.save(user);
  }
}
```

Código 16: Implementação para uso em produção

Podemos dizer que o ISP é respeitado na relação entre esses três módulos pois a interface IUsersRepository tem 100% dos seus métodos usados por suas implementações.

```
export default class FakeUsersRepository implements IUsersRepository {
  private users: User[] = [];

  public async findById(id: string): Promise<User | undefined> {
    return this.users.find(user => user.id === id);
  }

  public async findByEmail(email: string): Promise<User | undefined> {
    return this.users.find(user => user.email === email);
  }

  public async create(userData: ICreateUserDTO): Promise<User> {
    const user = new User();
    Object.assign(user, { id: uuidv4() }, userData);
    this.users.push(user);
    return user;
  }

  public async save(user: User): Promise<User> {
    const findIndex = this.users.findIndex(findUser => findUser.id ===
user.id);
    this.users[findIndex] = user;
    return user;
  }
}
```

Código 17: Implementação para uso em testes unitários

Na Figura 8, temos a representação gráfica de como os módulos citados se relacionam e como eles constituem o ISP. As setas tracejadas representam a implementação da interface IUsersRepository.

Esse princípio pode ser quebrado se, como exemplificado na Seção 2.2.4, a interface especificar um método que uma de suas implementações não necessite. Ao fazer isso, perde-se os benefícios que uma abstração de interfaces pode promover, como o reuso do código.

O benefício de se usar esse princípio está na desacoplação do código. Sem esse princípio aplicado, por exemplo, seria necessário especificar o subtipo UsersRepository, que, como demonstrado previamente, por se tratar de um Adaptador de Interfaces, é um módulo de nível inferior aos Casos de Uso.

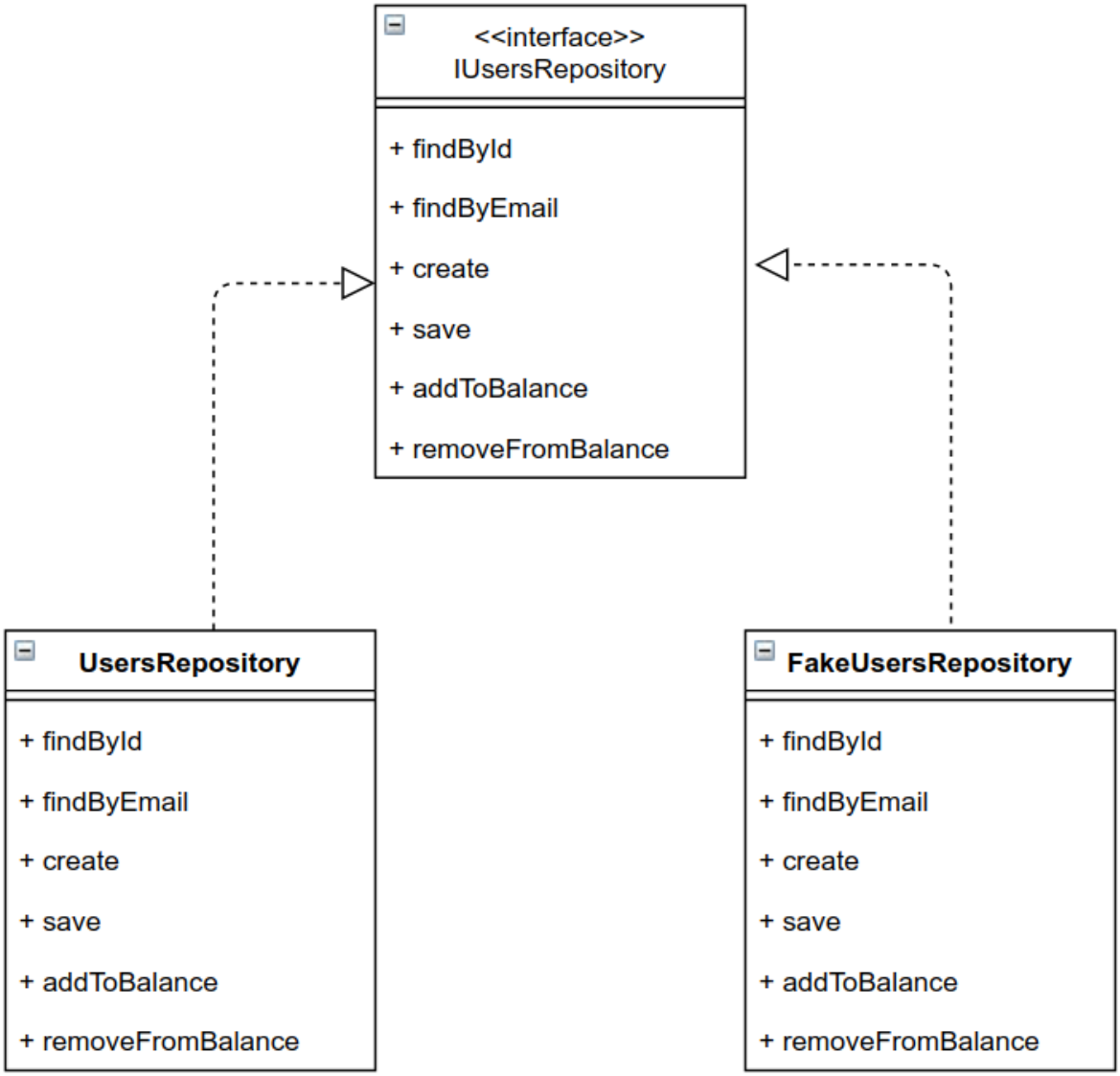

Figura 8: Representação do ISP

5.2.5 IMPLEMENTAÇÃO DO DIP

A aplicação do Princípio da Inversão de Dependência pode ser encontrada em todos os Casos de Uso da aplicação que interajam com o banco de dados,

visto que todos eles importam a dependência a partir de uma abstração. A Figura 9 ilustra essa situação com classes que já foram apresentadas previamente.

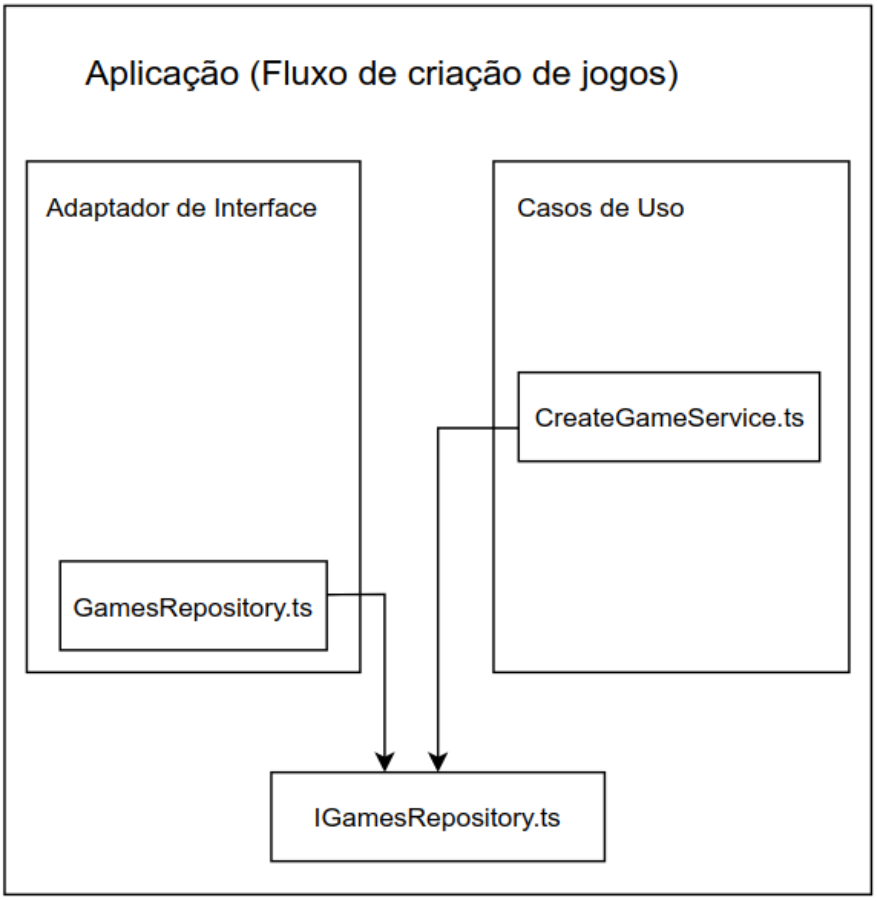

Figura 9: Representação do DIP

Para visualizar um modelo que viole a implementação do DIP, basta fazer que o Caso de Uso CreateGameService importe diretamente a dependência GamesRepository, como exemplificado através do Código 18 e da Figura 10.

```
export default class CreateGameService {
  constructor(
    private gamesRepository: new GamesRepository(),
  ) {}

  public async execute(data: ICreateGameDTO): Promise<Game> {
    //validacoes
    return await this.gamesRepository.create(/* dados do jogo */);
  }
}
```

Código 18: Violação do DIP

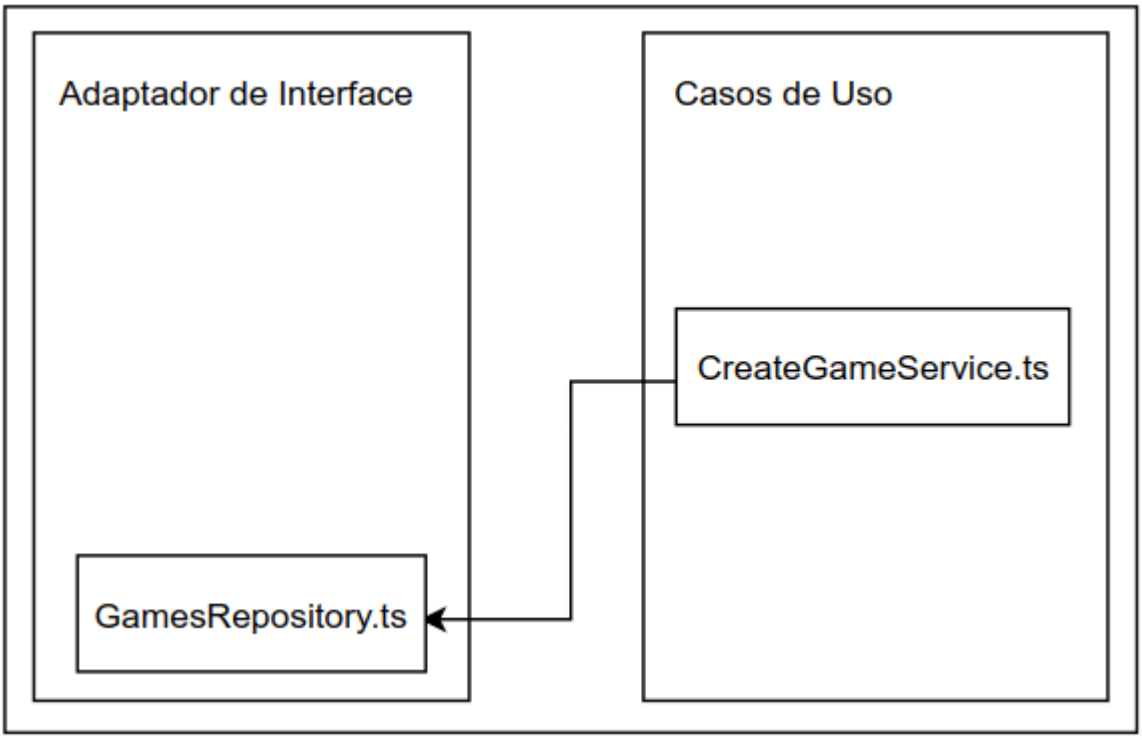

Figura 10: Violação do DIP

5.2.6 BENEFÍCIOS DO SOLID

Com isso, podemos refletir sobre os benefícios da aplicação desses princípios. O SRP e o OCP, por exemplo, garantem a facilidade de leitura e entendimento do código ao separar funcionalidades complexas em componentes menores.

O LSP e o ISP exploram e determinam regras para o uso do conceito de interface e herança de forma a garantir a coesão do código e que boas abstrações sejam construídas.

O DIP permite que módulos de nível superior possam referenciar, indiretamente, módulos de nível inferior sem acarretar acoplamento entre estes.

Por fim, se atentar aos princípios SOLID traz como principais benefício, de diferentes formas, um código mais manuseável e limpo, menos suscetível a bugs inesperados e com maior potencial de crescimento.

CONCLUSÕES E TRABALHOS FUTUROS

Por necessitar de um cuidado maior com certos detalhes, o desenvolvimento de uma aplicação que respeite aos princípios da *Clean Architecture* e do *SOLID* pode demandar mais tempo para ser concluído, principalmente por desenvolvedores inexperientes.

Isto porque as técnicas abordadas neste trabalho, conforme explicitado nos capítulos anteriores, estão muito mais relacionadas a um ganho futuro do que a resultados imediatos. Dito isto, apesar deste trabalho ser sido concluído em apenas quatro meses, já foi possível perceber os benefícios durante a construção do software deste projeto.

Por vezes, durante o desenvolvimento da aplicação web, foi necessário revisitar fluxos mais antigos e, graças a um bom delineamento dos limites entre as camadas da aplicação, foi possível realizar alterações pontuais sem que isso acarretasse em grandes mudanças para as camadas vizinhas.

Esta é a manuseabilidade garantida no uso dessas práticas e, para o desenvolvedor, pode ser considerado uma espécie de bom “investimento” dedicar um tempo extra para implementar essas práticas e mantê-las visto que, principalmente em softwares em constante evolução, é garantido que fluxos antigos tenham de sofrer manutenção, seja devido a bugs ou a novos recursos.

Porém, é muito importante que o desenvolvedor consiga entender quais são as prioridades para o projeto desenvolvido. Aplicações são desenvolvidas, geralmente, com o propósito de gerar lucro (através de novos produtos ou serviços) ou cortar gastos e agilizar demandas (com automatização de tarefas), e, para que esses objetivos sejam alcançados e as partes interessadas estejam satisfeitas, fundamentalmente, basta apenas que o código funcione.

Partindo da máxima que “Tempo é dinheiro”, podemos inferir que, para a maioria dos empregadores ou investidores que pouco conhece tecnologia, pouco

importa se produto será desenvolvido em Java ou TypeScript, se as tecnologias empregadas são as mais recentes ou se serão utilizadas boas práticas de desenvolvimento. A prioridade é que o produto desenvolvido seja entregue no prazo estimado.

Por isso, apesar das técnicas mencionadas serem efetivas para evitar o retrabalho, ao se deparar com prazos apertados, o desenvolvedor provavelmente optará por escolher a solução mais imediata e alguém, no futuro, irá pagar o preço por essa decisão.

Da mesma forma, projetos de software que requererão um menor tempo desenvolvimento ou que provavelmente não receberão muitos recursos com frequência provavelmente não precisam de tanto cuidado assim, visto que o tempo empregado para verificar se os princípios estão sendo respeitados ou analisar como refatorar o código a fim de respeitá-los pode ser um investimento que não trará tanto retorno assim.

Finalmente, como um desenvolvedor atuante no mercado, posso afirmar que já desenvolvi muito código ruim, assim como, apesar da pouca experiência, também já tive oportunidade de dar manutenção em código ruim. Por isso, posso afirmar que um trabalho voltado para as boas práticas de construção de software contribuiu e muito para minha formação.

Pretendo continuar me aprimorando, conhecendo mais práticas e metodologias de desenvolvimento de software e, paralelamente, ganhando mais experiência como desenvolvedor.

REFERÊNCIAS BIBLIOGRÁFICAS

1. MARTIN, Robert Cecil. **Arquitetura Limpa: O guia do Artesão para Estrutura e Design de Software**. Alta Books, 2018.
2. Carnegie Mellon University. **Software Architecture.** <https://www.sei.cmu.edu/our-work/software-architecture/> Acesso em 07 nov. 2021.
3. MARTIN, Robert Cecil. **The Principles of OOD**. <http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod> Acesso em 07 nov. 2021.
4. MARTIN, Robert Cecil. **The Clean Architecture**. <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html> Acesso em 07 nov. 2021.
5. OpenJS Foundation. **NodeJS**. <https://nodejs.org/pt-br/about/> Acesso em 07 nov. 2021.
6. Stack Overflow. **2021 Developer Survey**. <https://insights.stackoverflow.com/survey/2021#most-loved-dreaded-and-wanted-webframe-love-dread> Acesso em 07 nov. 2021.
7. Microsoft. **TSyringe Repository**. <https://github.com/microsoft/tsyringe> Acesso em 07 nov. 2021.
8. Microsoft. **TypeScript for the New Programmer**. <https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html> Acesso em 07 nov. 2021.
9. PostgreSQL. **Why Use PostgreSQL?** <https://www.postgresql.org/about/> Acesso em 07 nov. 2021.
10. TypeORM. **TypeORM** <https://typeorm.io/#/> Acesso em 07 nov. 2021.
11. TypeORM. **What is the Data Mapper pattern?** <https://typeorm.io/#/active-record-data-mapper/what-is-the-data-mapper-pattern> Acesso em 07 nov. 2021.

12. VENNERS, Bill. **Orthogonality and the DRY Principle. A Conversation with Andy Hunt and Dave Thomas, Part II** < https://www.artima.com/articles/orthogonality-and-the-dry-principle> Acesso em 07 nov. 2021
13. Apache. **KISS Principle** <https://people.apache.org/~fhanik/kiss.html> Acesso em 19 nov. 2021
14. Ron Jeffries. **You’re NOT gonna need it!** <https://ronjeffries.com/xprog/articles/practices/pracnotneed/> Acesso em 19 nov. 2021.

ANEXO A – MODELOS DE DADOS

A.1 Modelo lógico de dados

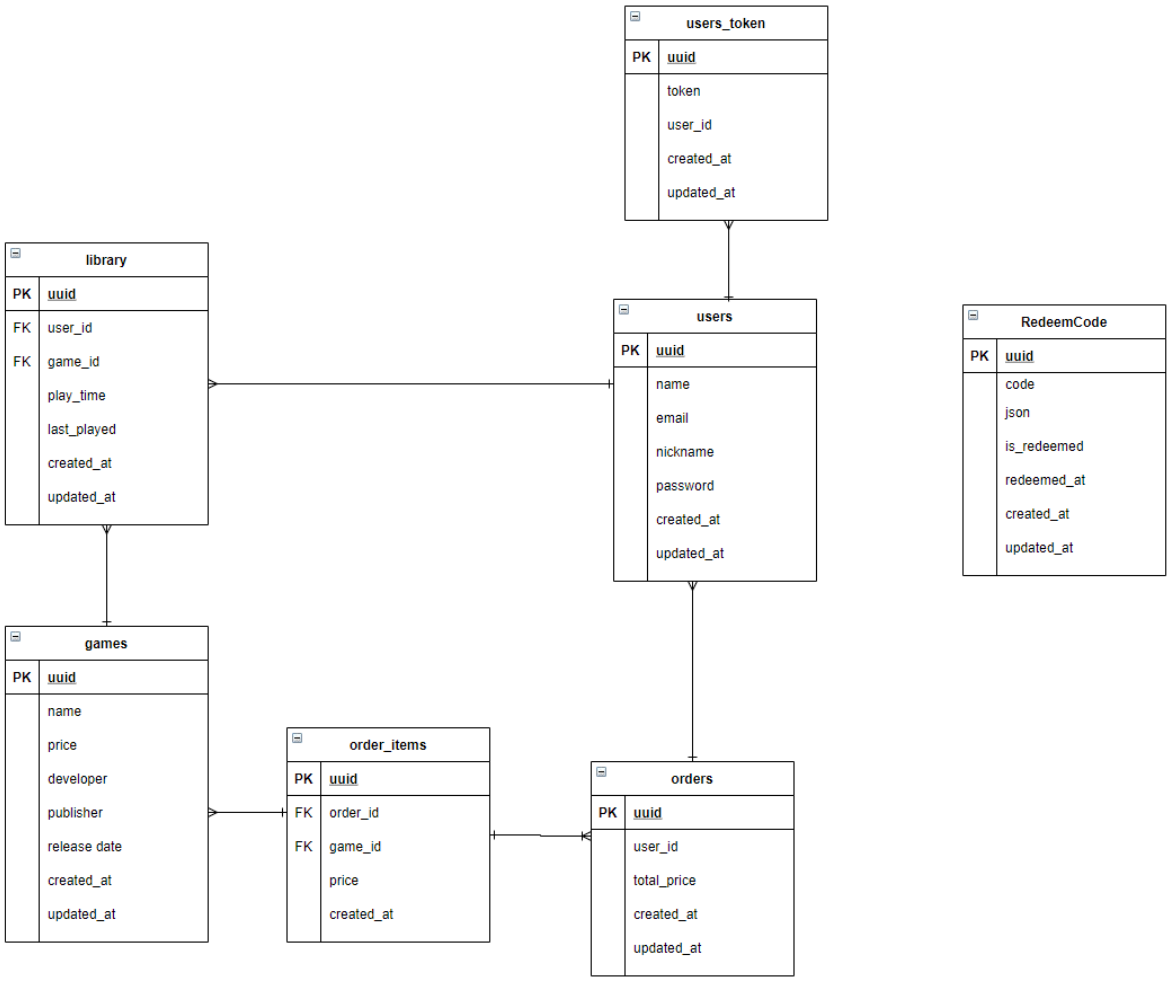

Figura 11: Modelo Lógico de Dados

ANEXO B – FLUXOGRAMAS DA APLICAÇÃO

B.1 Autenticação do Usuário

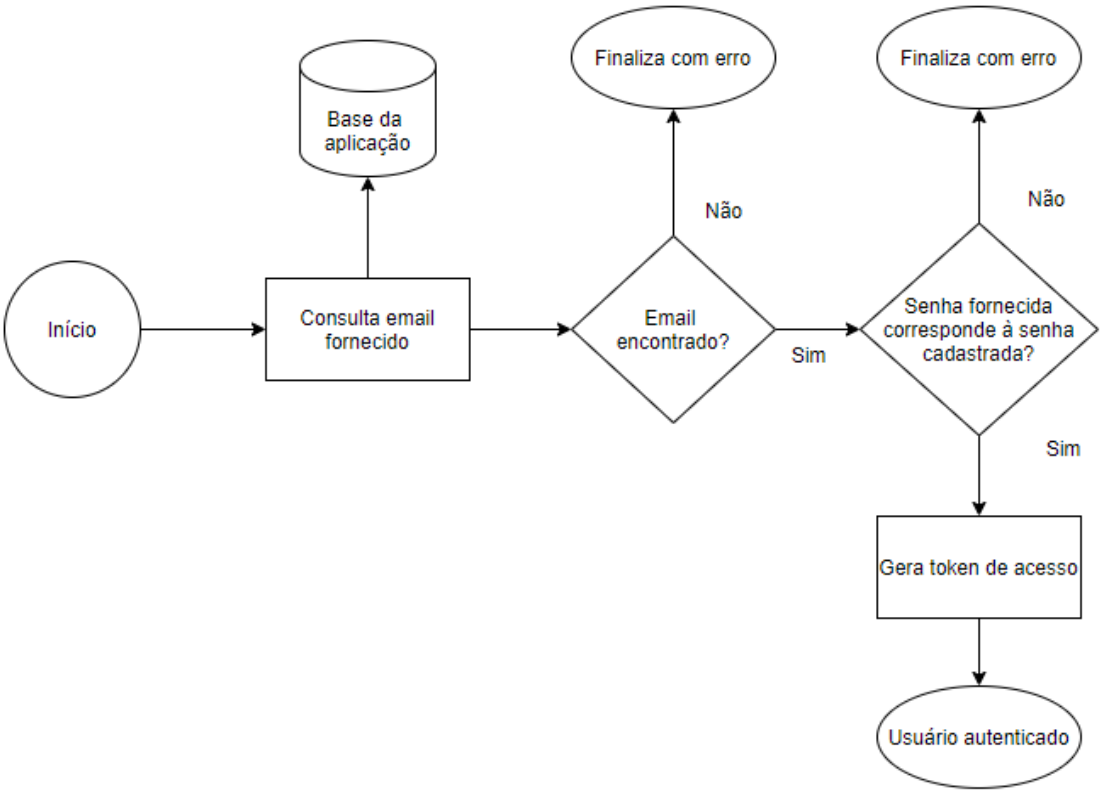

Figura 12: Fluxo de Autenticação do Usuário

B.2 Cadastro de Usuários

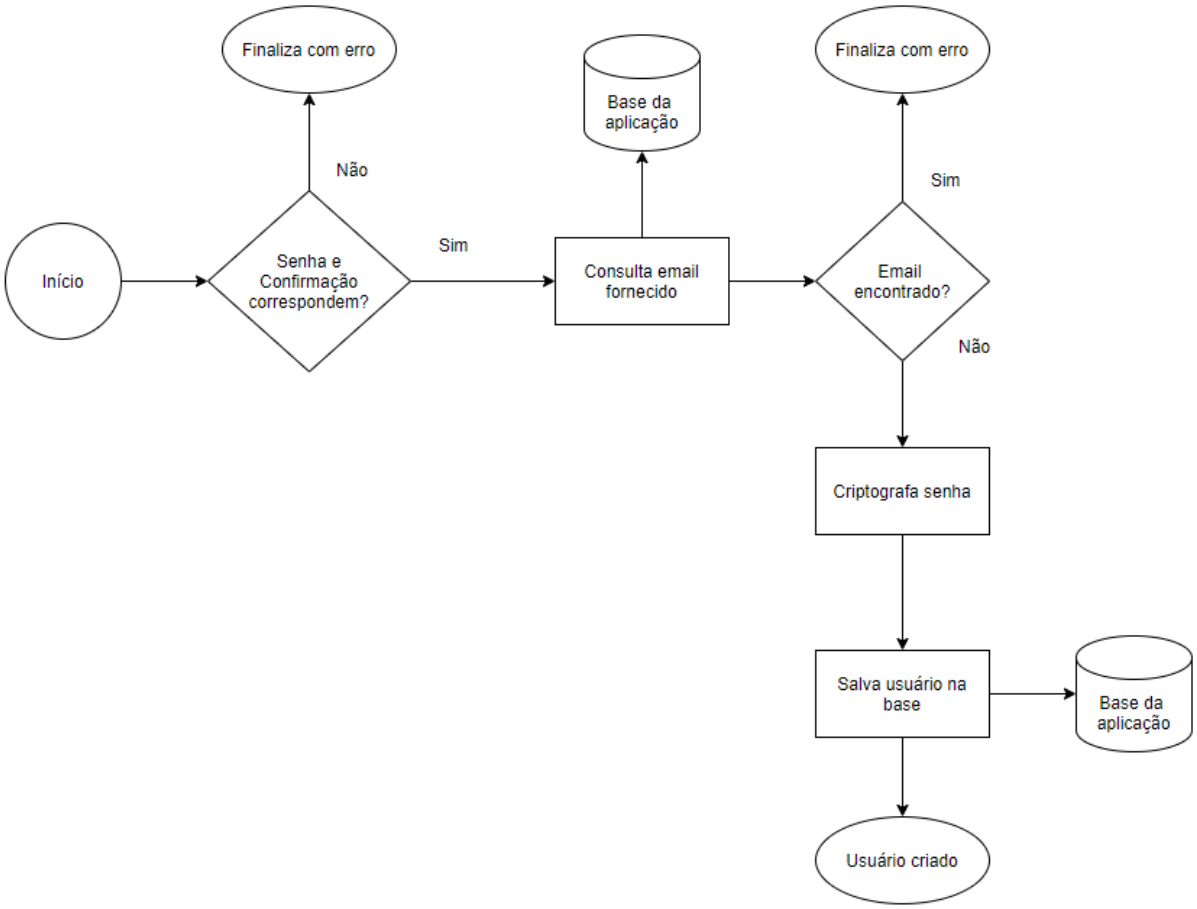

Figura 13: Fluxo de Cadastro de Usuários

B.3 Exibição de Usuários

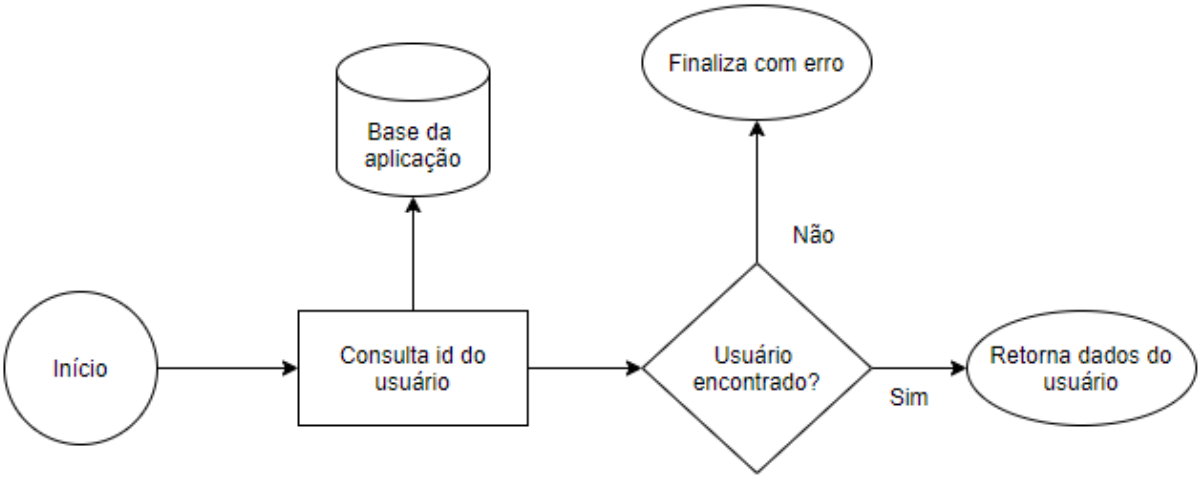

Figura 14: Fluxo de exibição de Usuários

B.4 Atualização de Usuários

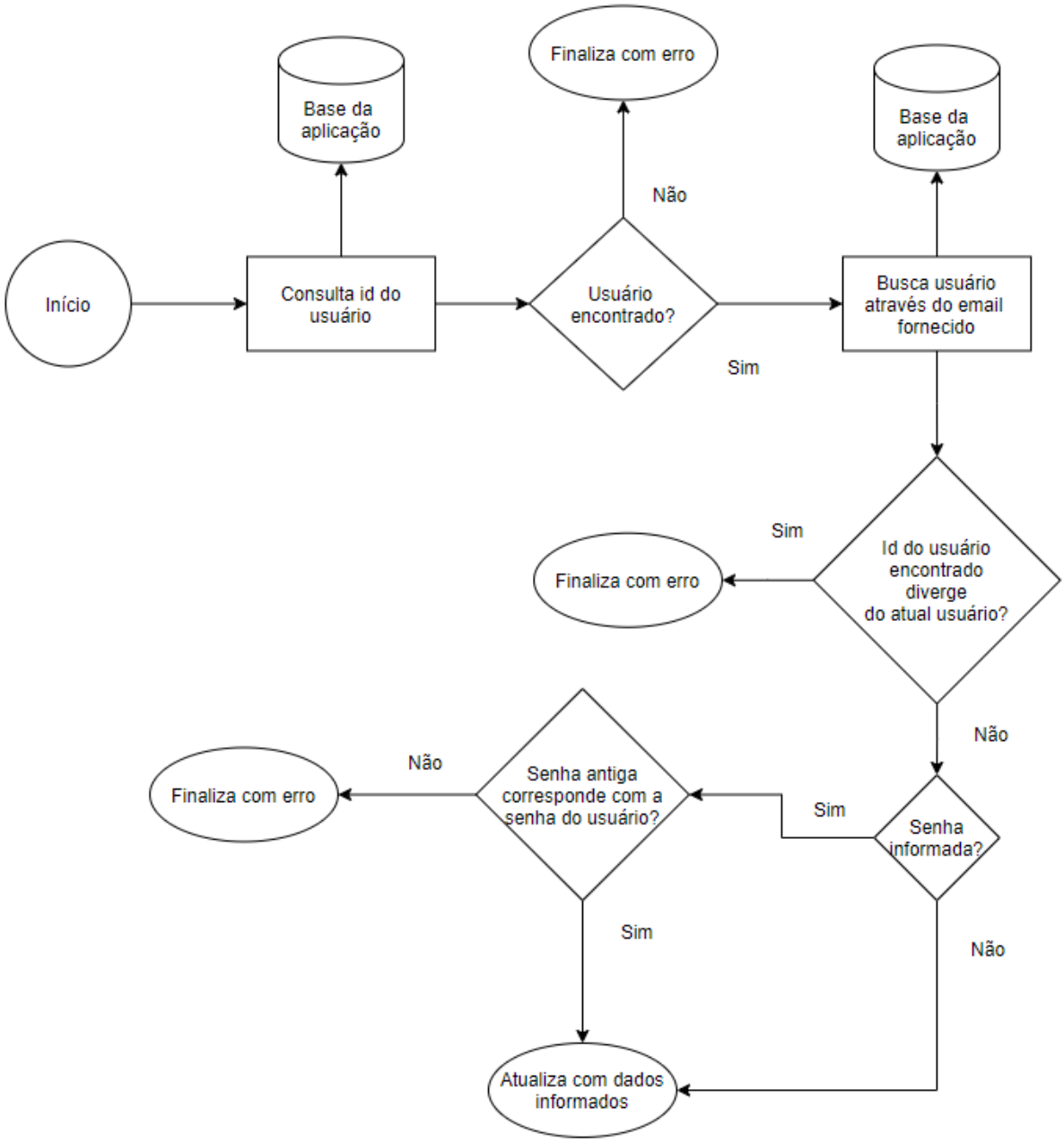

Figura 15: Fluxo de atualização de Usuário

B.5 Inserção de Jogos

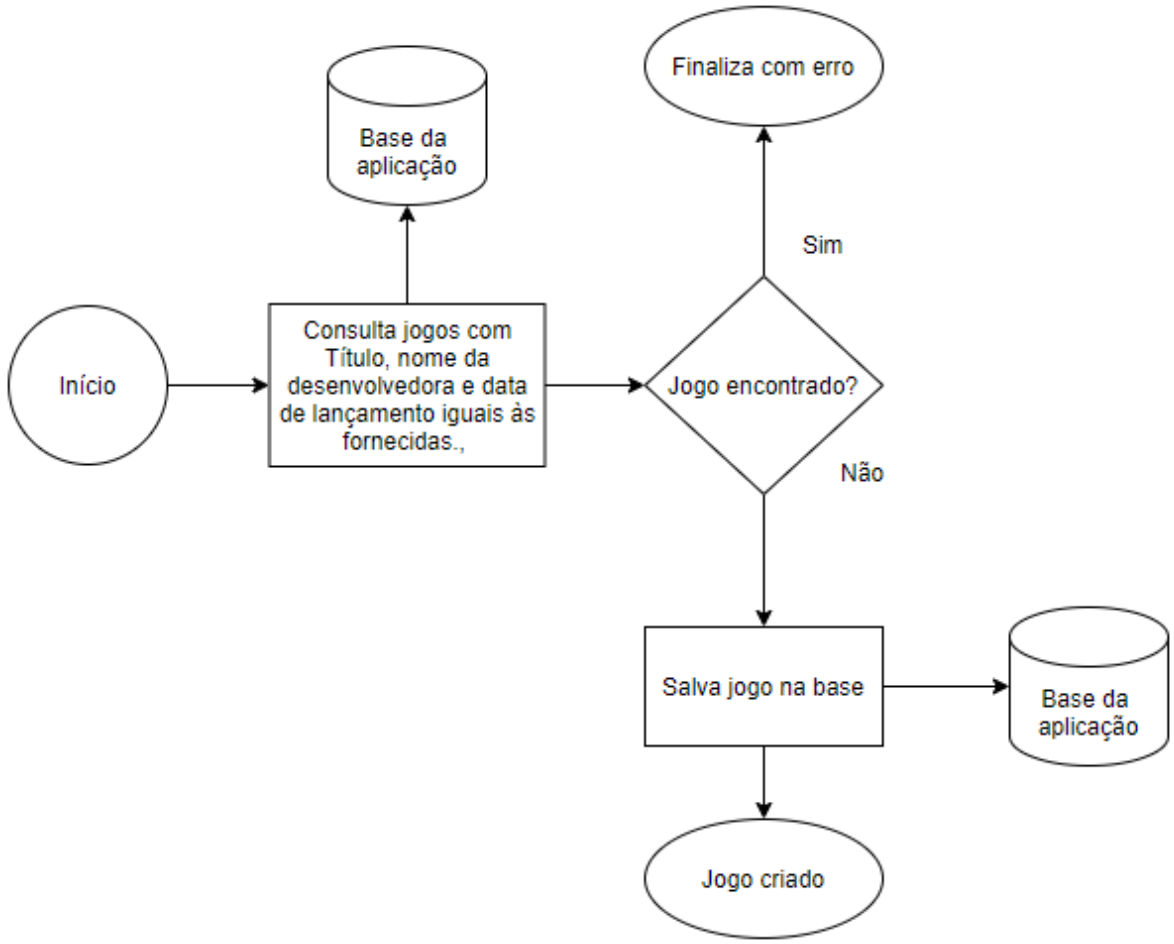

Figura 16: Fluxo de inserção de jogos

B.6 Atualização de um jogo

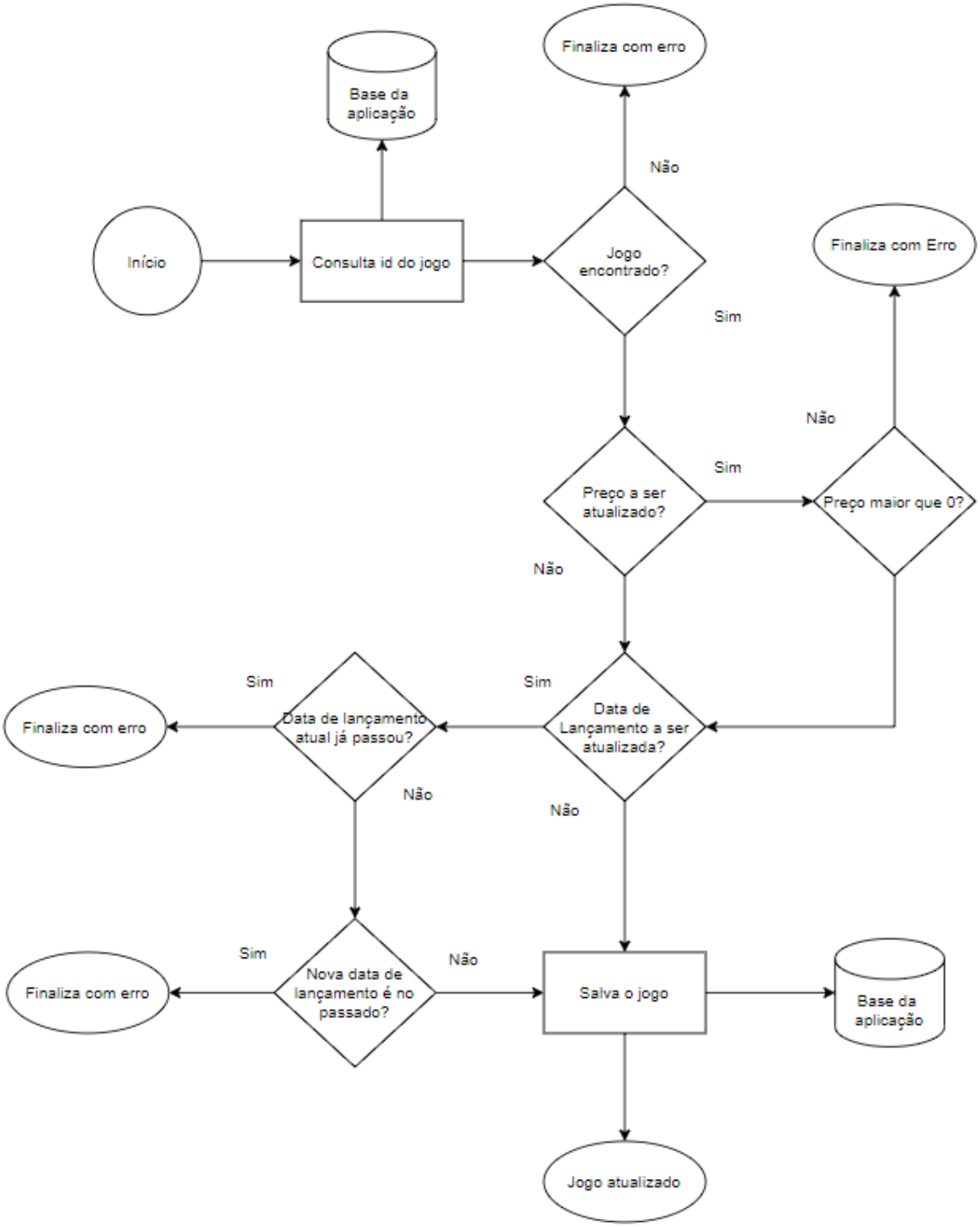

Figura 17: Fluxo de atualização de um Jogo

B.7 Exibição de um jogo

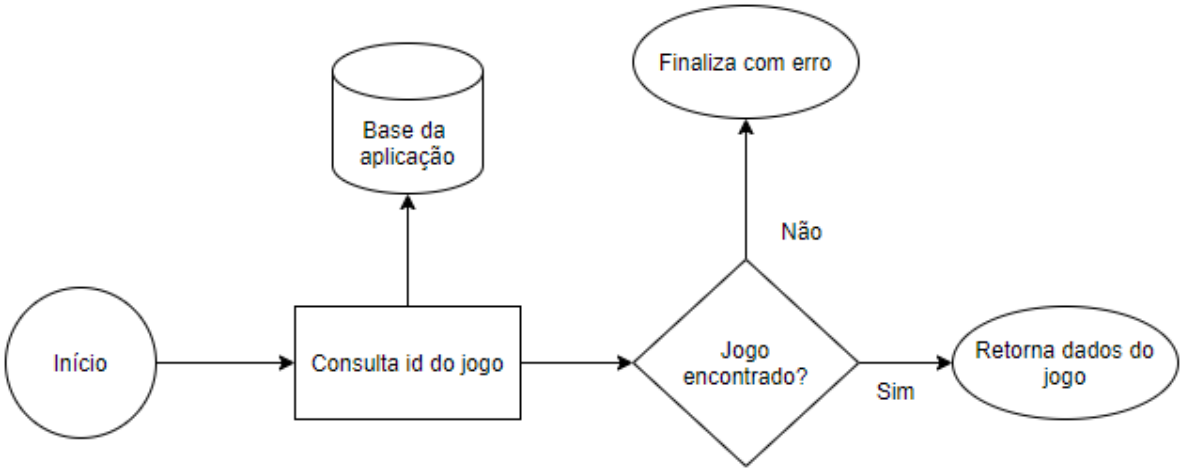

Figura 18: Fluxo de exibição de jogos

B.8 Listagem de jogos

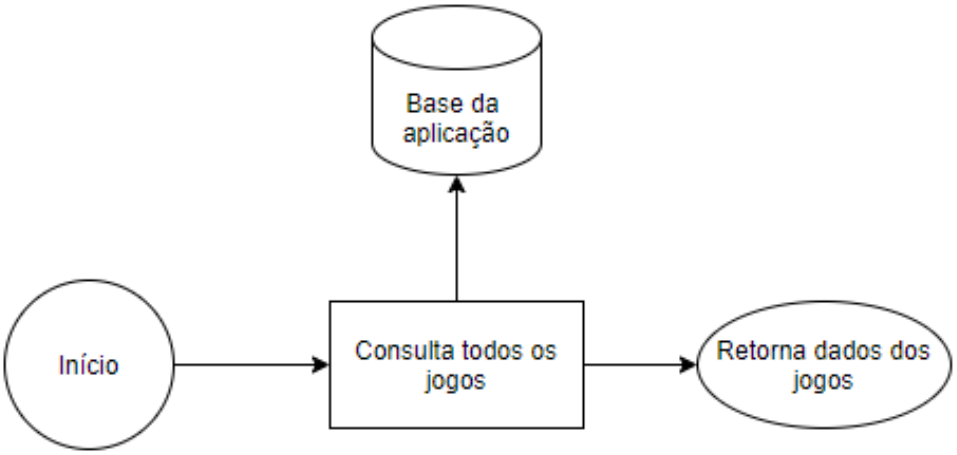

Figura 19: Fluxo de Listagem de jogos

B.9 Criação de códigos de reivindicação de dinheiro virtual

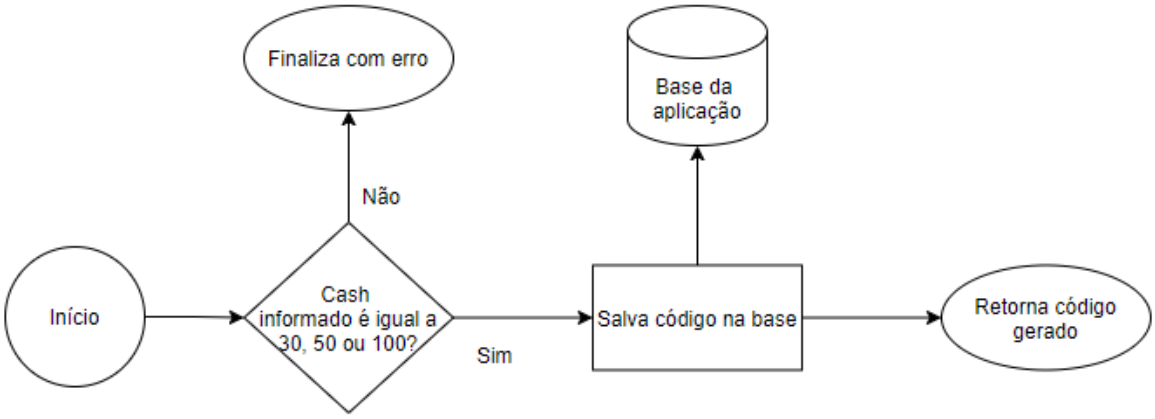

Figura 20: Fluxo de criação de códigos de *cash*

B.10 Criação de códigos de reivindicação de jogos

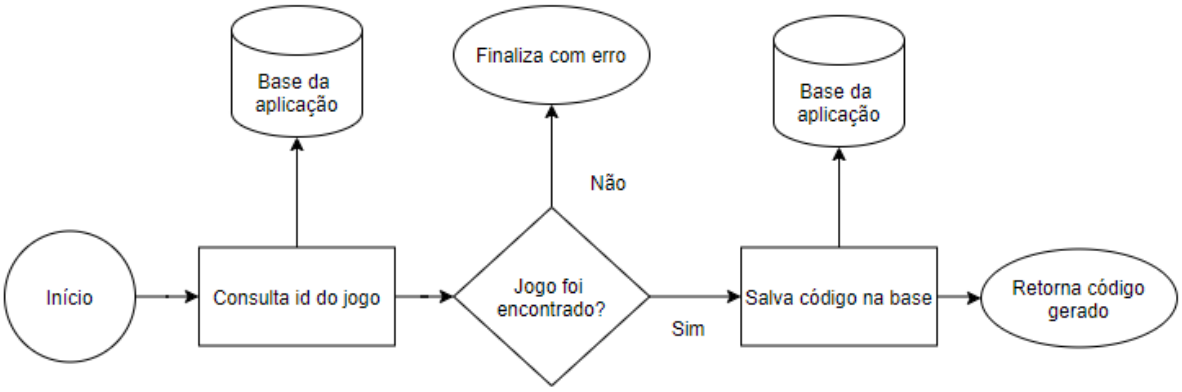

Figura 21: Fluxo de criação de códigos de jogo

B. 11 Reivindicação de jogos

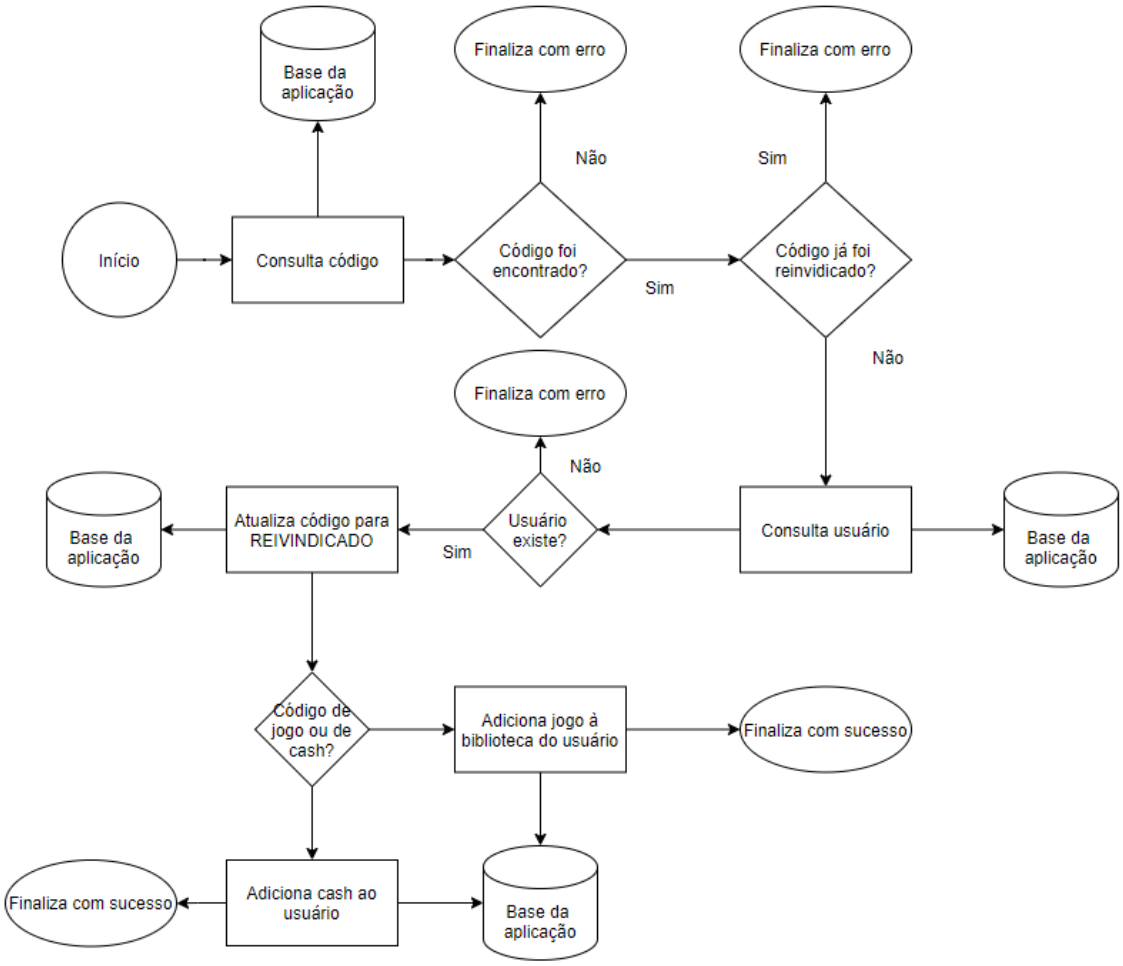

Figura 22: Fluxo de Reivindicação de Códigos

B.12 Compra de jogos

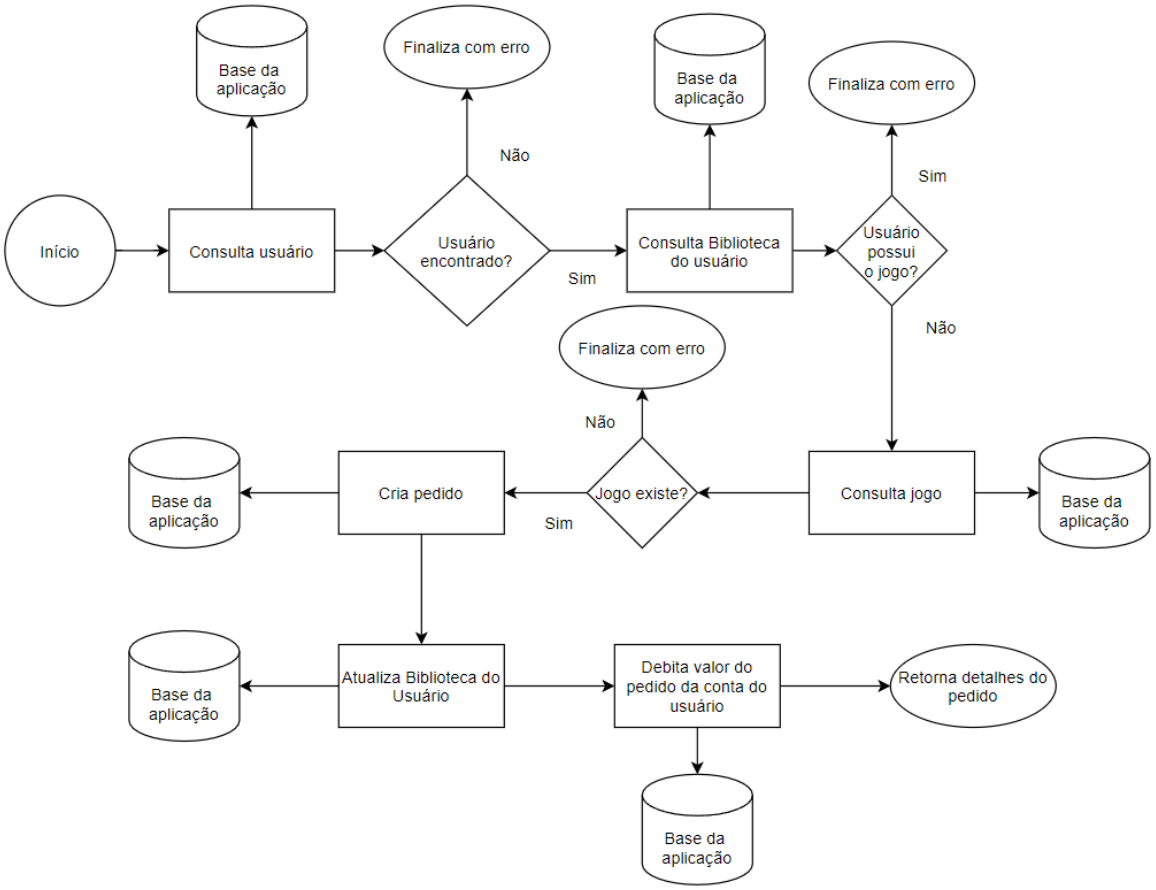

Figura 23: Fluxo de compra de jogos

B.13 Listar Compras

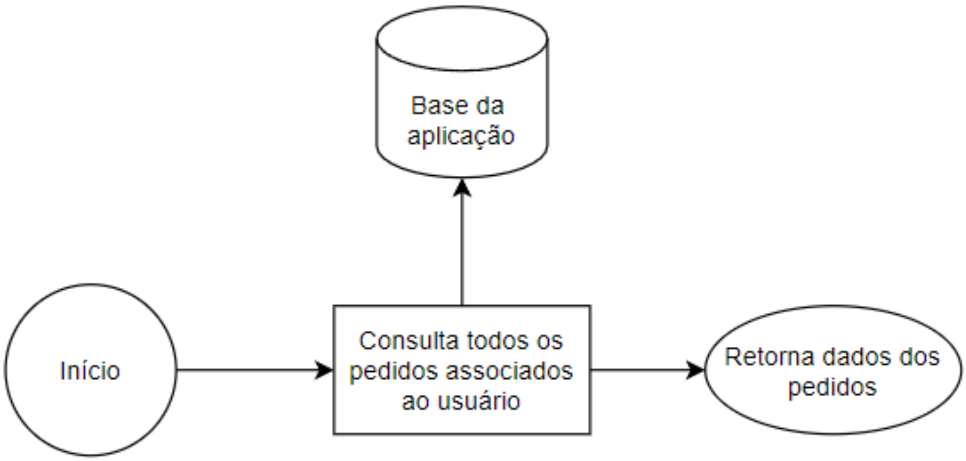

Figura 24: Fluxo de exibição de histórico de compras

B.14 Exibir lista de jogos adquiridos

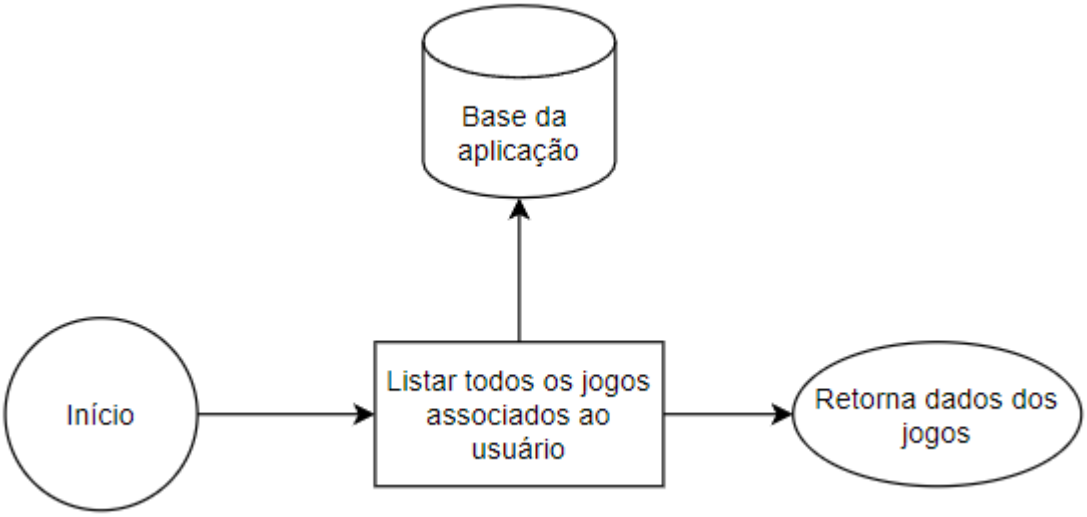

Figura 25: Fluxo de exibição de lista de jogos adquiridos