Arquitetura de *software* aplicada ao front-end

Alessandro Rodrigo F. P. Junior
Fundação Educacional de Fernandópolis (FEF)
Fernandópolis, Brasil
alessandro.fresneda84@gmail.com

Jefferson Antonio Ribeiro Passerini
Fundação Educacional de Fernandópolis (FEF)
Fernandópolis, Brasil
jefferson.passerini@fef.edu.br

***Resumo*—Este trabalho tem por objeto o estudo de arquiteturas de software e aplicação da arquitetura conhecida como *Clean Architecture* no ecossistema do front-end, com a finalidade de tornar o projeto mais escalável, de fácil manutenibilidade, possibilitando a modelagem direta das regras de negócios, casos de uso e adaptadores, sem que o *framework* modele a aplicação. A arquitetura supracitada será aplicada à um projeto pré-existente, como forma de estudo. Este que contará com a divisão desse código nas camadas referentes à essa estrutura, desacoplamento através de aplicações de padrões de projeto como *Adapter* e *Dependency Injection* e princípios como *SOLID* e *DIP*. A finalidade do estudo é provar que através da modelagem da *Clean Architecture* em um projeto front-end, é possível trazer melhorias como: escalabilidade, manutenibilidade, testabilidade e menor curva de aprendizado para novos desenvolvedores do time.**

***Palavras Chave*—*Clean Architecture*. *Onion Architecture*. *Hexagonal Architecture*. Escalabilidade. Manutenibilidade. Modelagem de software.**

I. Introdução

No final da década de 60, a área da tecnologia enfrentou um período cujo o termo ficou conhecido como crise do software. Durante o próprio, o mercado como um todo enfrentou dificuldades em relação ao desenvolvimento de software devido a crescente demanda pelos mesmos. Porém, o problema não terminava nesse ponto, a falta de técnicas, padrões e planejamento aumentava drasticamente os principais problemas que eram: projetos além do orçamento, fora do prazo, baixa qualidade que muitas vezes não satisfaziam os requisitos e códigos difíceis de manter.

Ao longo dos anos 70, surgiram estudos acerca do assunto com o objetivo de sanar esses principais problemas através de ferramentas, padrões e técnicas. Isso deu-se início com Edsger Dijkstra, uma das figuras mais influentes na área de ciência da computação de sua época, ele foi um dos responsáveis por trás da aceitação da programação como disciplina científica. O próprio, em conjunto com um pequeno grupo de acadêmicos e programadores que participava, defendiam um novo estilo de programação, afim de solucionar os problemas pelos quais passavam. Desta forma, deu-se origem ao paradigma de programação conhecido como ”programação estruturada”. Com essa nova metodologia os profissionais da época foram capazes de gerenciar projetos de software cada vez mais complexos.

Após algum tempo com esse novo paradigma no mercado, as discussões cessaram, Dijkstra havia vencido. Conforme as linguagens evoluíram, as declarações *goto* desapareceram e a programação estruturada ganhou mais espaço. Hoje, comumente, ela é utilizada nas linguagens mais modernas, estas que não permitem o uso de uma indisciplinada transferência direta de controle [1].

De acordo com o supracitado, entende-se que esses paradigmas e, futuramente, arquiteturas completas nasceram da necessidade de organizar e escalar projetos na área de programação conforme a demanda crescia exponencialmente e de forma proporcional à complexidade do código. De acordo com os fatos, entende-se que a arquitetura de software de um sistema abrange a forma como suas partes são organizadas, incluindo questões como o comportamento dessa estrutura e quais componentes são responsáveis por realizar um conjunto específico de funções. Resumidamente, é uma estrutura padronizada, fundamentada em conceitos, reproduzível, sob o qual um sistema pode ser desenvolvido [2].

A escolha da arquitetura influencia aspectos como a produtividade da equipe, a qualidade do produto, facilidade de manutenção, testabilidade e escalabilidade [3]. Desta forma, essa decisão tem grande impacto no sucesso do projeto, principalmente a longo prazo. Hoje, existem diversos princípios e padrões que são utilizados nos sistemas e normalmente os projetos desenvolvidos não se limitam a um único estilo ou arquitetura. Em vez disso, são uma combinação de padrões que, juntos, formam o sistema completo.

Uma boa analogia é a arquitetura de uma casa e sua arquitetura. Faz parte da sua arquitetura sua forma, a aparência externa, as elevações e o layout dos espaços e salas, isso em um contexto amplo. Quando analisa-se as plantas do arquiteto pode-se ter acesso à um número imenso de detalhes de mais baixo nível. Informações como tomadas, interruptores, lâmpadas, quais interruptores controlam quais lâmpadas, onde cada elemento vai ser instalado, quanto espaço ele requer, suas fundações e estrutura de paredes em si. Em suma, os pequenos detalhes servem de base para as decisões de alto nível. Dado esse fato entende-se que o baixo nível e alto nível fazem parte do design da casa como um todo [1].

Este trabalho terá como objeto de estudo a aplicação da *Clean Architecture*, uma arquitetura conhecida por fazer parte do grupo de arquiteturas de *adapters*, que segrega sua modelagem em camadas e permite a comunicação com agentes externos através de adaptadores e vem ganhando força no mercado, em um projeto real. A finalidade desse estudo é entender de quais formas essa estrutura é benéfica para o projeto e provar seus principais pontos fortes, que foram pré-

estabelecidos pelo seu criador, Robert C. Martin [4].

II. Fundamentação Teórica

Este capítulo tem como objetivo apresentar os conceitos necessários para a elucidação da metodologia proposta, informações sobre padrões de projeto *Design Patterns* como: *Adapters*, *Hexagonal Architecture*, *Clean Architecture*.

A. Design Patterns

Design patterns ou padrões de projeto, em tradução literal, é um termo muito conhecido no campo de alto nível da programação, existem inúmeros padrões conhecidos, os mais comuns, são os presentes no catálogo do GoF (*Gang of Four*) [5]. A sua única responsabilidade é servir como uma solução típica e padronizada para problemas genéricos que acontecem em praticamente todos os projetos de uma forma ou outra [6].

Em contra partida, não é tão fácil quanto parece, é necessário uma análise profunda e elevado conhecimento técnico para saber como modelar o padrão e encaixá-lo de forma correta no código, pois tudo o que é servido ao profissional, é a teoria. Eles consistem basicamente em três principais pontos: Propósito, motivação e estruturas.

Os padrões são divididos em categorias, são elas: Criacionais, estruturais e comportamentais. Eles tem por objetivo, fornecer formas e meios flexíveis e reutilizáveis de criação de objetos, flexibilidade e eficiência de estruturas maiores, em conjunto com classes e objetos e gerenciar comunicação e assinalação de responsabilidades dentro do código, respectivamente [6].

B. Adapters

O termo *Adapters* é conhecido no meio por ser um *Design Pattern* da categoria estrutural [7], portanto seu objetivo dentro do código é manter estruturas grandes flexíveis e eficientes. Ele mostra-se extremamente útil em cenários onde precisa-se consumir um ou vários serviços pré-existentes, mas que não fornecem a interface que o software necessita [6].

O funcionamento desse padrão consiste basicamente em servir como intermediário de um serviço, biblioteca ou agente externo que seja incompatível com a interface do sistema, ele atua na implementação concreta de uma interface para oferecer ao software uma forma padronizada de consumir um ou mais serviços de mesma finalidade. Essa abordagem é importantíssima para entender o conceito principal das arquiteturas de *adapters*, na (Fig.1) pode-ser ver uma adaptação gráfica de como funciona esse padrão [6].

C. Hexagonal Architecture ou Ports and Adapters

A (Fig.2) é uma representação da *Hexagonal Architecture* [9], ela é amplamente conhecida e recomendada quando trata-se de arquiteturas baseadas em adaptadores. Sua representação gráfica não dita o limite máximo de conexões, podendo haver mais elementos do que o número de arestas de um hexágono.

Esse modelo arquitetural faz parte de um grupo que vem ganhando muita força no mercado de desenvolvimento de software e compartilha dos mesmos princípios, como *Clean Architecture*, *Onion Architecture* e a própria *Hexagonal Architecture* [10], como Martin comenta em seu livro [1], esses modelos emergentes tem como objetivo principal segregar a aplicação em camadas, ser independentes de *frameworks*, interfaces de usuário, banco de dados e qualquer outro tipo de agente externo e serem altamente testáveis. Todas elas, tem uma camada responsável por disponibilizar interfaces a serem seguidas por seus módulos externos, isso gera a facilidade em utilizar diferentes agentes externos de forma rápida e padronizada.

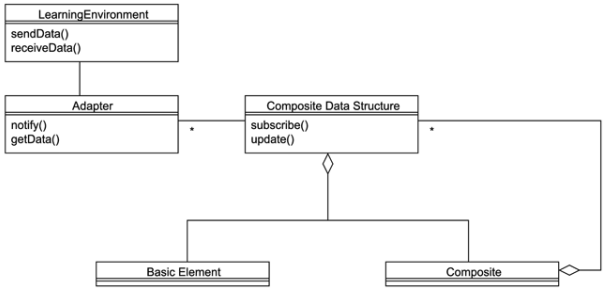


Fig. 1. Representação estrutural de um *Adapter Pattern* [8]

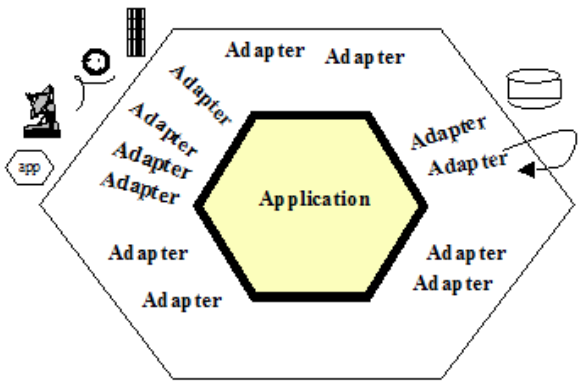


Fig. 2. Diagrama de *Hexagonal Architecture* [9]

Em suma, falando das arquiteturas que compõem esse grupo, espera-se que todas tenham camadas, adaptadores, núcleos de negócios e componentes pouco acoplados. Tudo isso em conjunto faz com o que o projeto torne-se fácil de manter, escalar e testar.

D. Clean Architecture

A arquitetura limpa, como aludido, em uma tradução literal, foi criada em meados de 2012 por Robert Cecil Martin, conhecido na comunidade de TI como "*Uncle Bob*" [4], ela, em conjunto com a *Onion Architecture* [11], são derivadas da *Ports and Adapters*, portanto é mais que lógico que compartilhem os mesmos fundamentos. Ela permite ao arquiteto modelar as regras de negócio sem interferência externa, isso por que nessa arquitetura o *core* da aplicação não enxerga o "mundo externo", isso garante mais clareza, desacoplamento, manutenibilidade e escalabilidade.

Como pode-se ver na (Fig.3), um diagrama de modelo ideal da arquitetura limpa, ela possui camadas muito bem definidas e seu núcleo é composto por suas entidades e *use cases* que em conjunto de suas camadas mais internas compõem seu domínio. Em suma, quanto mais próximo do centro, mais próximo das regras de negócio. O diagrama deixa claro, através do seu direcionamento de *Dependency Rule*, regra de dependência, em tradução literal, que os níveis externos devem depender dos níveis internos, isso traz autonomia para o núcleo da aplicação, que não precisa se preocupar com o que acontece nas camadas mais externas. Os adaptadores que fazem o trabalho de permitir e padronizar o acesso das camadas externas às camadas mais internas, também são fortes responsáveis por essa independência [1], eles permitem que a aplicação trabalhe com qualquer agente externo, como bibliotecas, *frameworks* e afins, no estilo *plug-in*, pois uma vez que os adaptadores fornecem uma interface, o programador precisa apenas ajustar o funcionamento do agente externo. Essa será a arquitetura aplicada nesse trabalho.

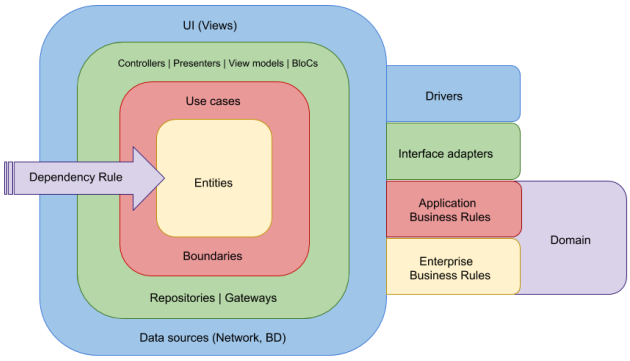


Fig. 3. Diagrama de *Clean Architecture*

E. Princípios SOLID

Esse conjunto de princípios denominado **SOLID** é um acrônimo de (*Single Responsability Principle*, *Open-Closed Principle* [12], *Liskov Substitution Principle* [13], *Interface Segregation Principle* e *Dependency Inversion Principle*) tem como finalidade, manter o código segregado, desacoplado, de fácil manutenibilidade e testável, seu foco é atuar em nível de código. Cada princípio tem um criador, um fundamento e um objetivo [14]. Ele foi introduzido por Robert C. Martin, em seu artigo [15].

F. Princípio da Inversão de Dependência

Esse é um termo conhecido por compor o acrônimo supracitado, *DIP*, de acordo com Martin, é o princípio que tem como objetivo que os módulos de alto nível não dependam dos de baixo nível dentro do código. Sua ideia é que os módulos não tenham dependências muito acopladas, ele resolve esse problema, fazendo com o que o próprio dependa de uma interface que vai ser implementada por qualquer módulo de baixo nível e utilizada nos módulos de alto nível, isso permite que o sistema continue funcionando mesmo sem o alto nível ter conhecimento exato do módulo baixo nível que está sendo utilizado [15].

G. Injeção de dependência

A injeção de dependência é diferente do princípio *DIP* do *SOLID*, ela é um padrão de projeto e seu melhor uso é feito quando o DIP está presente e sendo bem aplicado [16]. Seu fundamento baseia-se em permitir que módulos recebam em seu construtor ou parâmetro, de qualquer tipo, uma dependência externa, que siga uma interface e a partir dessa interface, o uso seja padronizado [17].

III. Trabalhos Correlatos

Na construção do sistema de organização de navios de contêineres, realizado por Chen, Chen, Zhang e Sui [18], o *Adapter Pattern* foi usado para o ganho de manutenibilidade, escalabilidade e reusabilidade com sucesso, onde apontaram que esse padrão pode ser utilizado de forma micro e macro dentro do código.

Rajam, Cortez, Vazhenin e Bhalla [19], aplicaram o conceito de *Dependency Injection* em uma arquitetura *MVC* [20], a partir de um sistema de *e-Learning* baseado em um modelo de gerenciamento de tarefas. Concluíram que o padrão *DI* auxiliou no desacoplamento, na modularidade, na autonomia e na reutilização.

Harrer, Pinkwart, McLaren e Scheuer [8] exploraram o *Adapter Pattern* para desenvolver uma estrutura arquitetônica reutilizável e escalável e obtiveram resultados satisfatórios em questão de reutilização e escalabilidade, graças ao mecanismo de controle fornecido em forma de um adaptador pelo padrão.

Boukhary e Colmenares [21] propuseram a utilização da *Clean Architecture* para modelagem de aplicações utilizando a linguagem *Flutter* onde fornece uma solução para o problema de gerenciamento de estado, bem como uma escolha geral para a arquitetura de aplicativo móvel Flutter.

Em outra abordagem Arango e Loaiza [22] propuseram uma extensão do *framework* SCRUM denominada SCRUM-CA utilizando das boas práticas da *Clean Architecture*. A proposta dos autores foi validada através de um projeto de implementação, onde os resultados demonstraram uma melhora na manutenibilidade do software do projeto reduzindo a redundância de código.

Sondha et al [23] demonstram a aplicação de das boas práticas da *Clean Architecture* na criação de um *framework* de geração de código para desenvolvimento móvel para dispositivos Android. Os testes demonstraram que o *framework* conseguiu economizar 42% do tempo e da carga no desenvolvimento de aplicações para dispositivos Android, assim como o código gerado obteve um excelente nível de manutenibilidade.

O trabalhos citados demonstram a versatilidade e a usabilidade de padrões de projeto podendo ser utilizadas para estruturar o desenvolvimento de um software web ou *mobile* comum, mas também é aplicado a reformulação de *frameworks* de gestão de desenvolvimento de software ou ainda a softwares geradores de softwares.

IV. Metodologia

Nesse trabalho, a *Clean Architecture* foi aplicada à um projeto desenvolvido com o *framework React Native*, utilizando a linguagem *Typescript*, que é um *superset* do *Javascript*. Para a realização e codificação do sistema, foi utilizado o sistema operacional *Linux*, distribuição conhecida como Pop OS, derivada do Ubuntu, em cojunto com um software de codificação chamado *Visual Studio Code*, oferecido livremente pela empresa *Microsoft*. Para testes e desenvolvimento do mesmo, foi utilizado o simulador de dispositivo *Android* oferecido pelo *software Android Studio* e um dispositivo *Android* real, modelo Xiaomi Redmi Note 8.

A ideia principal foi realizar o desenvolvimento de um *marketplace* que comercializa recursos provindos da área rural em forma de aplicativo móvel, no entanto, o projeto não é o cerne, tendo servido qualquer outra base de código elegível para aplicação da arquitetura. A aplicação da própria supracitada foi realizada separando as camadas entre ***data***, ***domain***, ***infra***, ***main*** e ***presentation*** e atribuindo os devidos elementos que compõem o sistema à elas.

O estudo foi feito em forma de demonstração técnica do código do sistema suprarreferido, afim de obter um cenário real para obtenção de resultados. Dado isto, as análises ocorreram a cerca da estruturação do código do sistema, com o propósito de obter manutenibilidade, reutilização e escalabilidade. Percebe-se esses pontos dentro do código, analisando a segregação dos componentes, o desacoplamento entre os mesmos e principalmente a disponibilização de interfaces em forma de adaptadores.

Fig. 4. Estrutura de pastas da *Clean Architecture*

Através da estrutura de pasta, da (Fig. 4), percebe-se as camadas, propostas pela arquitetura de Martin [1], dentro desse trabalho, será utilizado e estudado as seguintes camadas [24]:

A. Camada data

Esta que é a camada responsável por lidar com persistência e gerenciamento de dados da aplicação, comunicação com banco de dados, API's e qualquer serviço referente aos dados e sua persistência. De acordo com, Bui, a finalidade desta camada é separar a regra de negócio de como os dados são armazenados e gerenciados [25].

B. Camada domain

Principal camada de toda a estrutura, ela quem guarda todos os casos de uso, modelos e regras de uso da aplicação. Dentro da camada domínio, fica todo o núcleo da aplicação, seu *core business*, de qual todo o restante da arquitetura depende. Por se tratar do coração da aplicação, é possível aplicar metodologias como o DDD, que trabalha diretamente em cima dessa responsabilidade. Isso confirma-se em conjunto com a ideia de Boukhary, ele afirma que essa camada é responsável por guardar e gerenciar as entidades e casos de uso que compõem as regras de negócios [24].

C. Camada infra

Essa camada é responsável por concretizar a implementação dos drivers através dos adaptadores disponibilizados pelas outras camadas, ela constrói e mantém a infraestrutura concreta da aplicação e disponibiliza para as outras camadas esses agentes externos implementados de acordo com o contrato do adaptador. Como diz, Sena, essa é a camada mais externa, que comunica-se diretamente com os agentes externos [26].

D. Camada main

Lida com implementações de *design patterns*, configurações de rotas, serviços internos do projeto, bibliotecas de ambiente de desenvolvimento e tudo o que for interno, porém de use generalizado para o projeto.

E. Camada presentation

Camada responsável por lidar com as informações que serão apresentadas na interface do usuário, nosso *framework front-end*, React, Vue, Svelte, Angular e muitos outros, ficam segregados nessa camada. Basicamente consomem as outras camadas e apresentam essas informações na tela. Conclui-se, em conjunto com a ideia de Boukhary, que essa é a camada mais dependente do *framework*, pois é nele onde o próprio fica e se desenvolve [24].

V. Resultados e Discussão

Com o desenvolvimento de uma arquitetura como a *Clean Architecture* no *front-end*, pode-se ter um ambiente lógico, completamente voltado à modelagem de negócio, com fluxos, entidades, regras e casos de uso completamente desacoplados, testáveis e sem dependência direta de uma única ferramenta ou *framework*. Em suma, nasce a possibilidade de utilizar o mesmo núcleo de aplicação em diferentes bases, por exemplo, consumir da mesma fonte em diferentes aplicações [27], e manter em suas responsabilidades, apenas arquiteturas que lhes dizem respeito, como a arquitetura Flux [28] e seus *life cycles*. Para tal, é necessário uma estrutura propicia, como apresentado na (Fig.4)

A. Utilidade do princípio DIP e o padrão de injeção de dependência

Neste trabalho, a utilização desse princípio e desse padrão foi dada com a finalidade de otimizar pontos cruciais e corroborarem com os adaptadores, afim de otimizar o desacoplamento, manutenibilidade, testabilidade e escalabilidade. Por meio da aplicação do princípio *DIP*, alcança-se um cenário ideal para o padrão de injeção de dependência, que permite o sistema utilizar mais de uma dependência de forma unificada,

ou seja, sem modificações no código para cada uma delas. Mais detalhes e exemplos práticos desse ecossistema e sua implementação são citados logo abaixo.

B. Os adaptadores e a ideia de plug-in

```ts
export interface CacheClientAdapter {
  getItem(key: string): Promise<string | null>;
  setItem(key: string, value: unknown): Promise<void>;
  removeItem(key: string): Promise<void>;
}
```

Fig. 5. Exemplo de *adapter interface*

```ts
import AsyncStorage from '@react-native-async-storage/async-storage';
import { CacheClientAdapter } from '../adapters/CacheClientAdapter';

export default class AsyncStorageAdaptee implements CacheClientAdapter {
  async removeItem(key: string): Promise<void> {
    await AsyncStorage.removeItem(key);
  }

  async getItem(key: string): Promise<string | null> {
    return AsyncStorage.getItem(key);
  }

  async setItem(key: string, value: any): Promise<void> {
    await AsyncStorage.setItem(key, value);
  }
}
```

Fig. 6. Exemplo de implementação da interface da (Fig 5) utilizando Async Storage

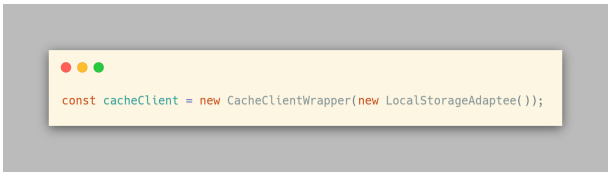
```ts
import { CacheClientAdapter } from '../adapters/CacheClientAdapter';

export default class LocalStorageAdaptee implements CacheClientAdapter {
  async removeItem(key: string): Promise<void> {
    await localStorage.removeItem(key);
  }

  async getItem(key: string): Promise<string | null> {
    return localStorage.getItem(key);
  }

  async setItem(key: string, value: any): Promise<void> {
    await localStorage.setItem(key, value);
  }
}
```

Fig. 7. Exemplo de implementação da interface da (Fig 5) utilizando Local Storage

A (Fig.5) apresenta uma interface nomeada de *CacheClientAdapter*, a finalidade dela é servir como base para o padrão *Adapter* [7] e permitir que o sistema utilize de forma padronizada, um ou vários gerenciadores de cache, sem afetar outras partes do código. Seu propósito é ser implementada por uma classe concreta, como é exemplificado na (Fig 6) e (Fig. 7).

```ts
import AsyncStorageAdaptee from '../adaptees/cacheClient/AsyncStorageAdaptee';
import { CacheClientAdapter } from '../../data/adapters/CacheClientAdapter';

export default class CacheClientWrapper implements CacheClientAdapter {
  constructor(
    private cacheClient: CacheClientAdapter = new AsyncStorageAdaptee(),
  ) {
    this.cacheClient = cacheClient;
  }

  async removeItem(key: string): Promise<void> {
    await this.cacheClient.removeItem(key);
  }

  async getItem(key: string): Promise<string | null> {
    return this.cacheClient.getItem(key);
  }

  async setItem(key: string, value: any): Promise<void> {
    await this.cacheClient.setItem(key, value);
  }
}
```

Fig. 8. Implementação de *wrapper* para controle unificado de *driver* externo



```ts
const cacheClient = new CacheClientWrapper(new LocalStorageAdaptee());
```

Fig. 9. Uso real do *wrapper* apresentado na (Fig. 8), com injeção de dependência

Quando essa interface é implementada por uma ou mais classes concretas, é possível utilizá-las de forma singular, pois seguem a mesma interface, com isso, os módulos que dependem delas não precisam conhecer detalhes de sua implementação. Desta forma, em conjunto com o padrão *Adapter*, responsável por oferecer uma interface para os *drivers* externos se ”adaptarem” ao funcionamento que o sistema espera, com o princípio *DIP* do conjunto de princípios *SOLID* que oferece ao software um modelo ideal de desacoplamento de dependências e por fim com o padrão de injeção de dependência, onde podemos injetar, externamente, a dependência do módulo, obtêm-se um cenário desacoplado, testável, manutenível e escalável.

A implementação do *DIP* e da injeção de dependência acontecem no *wrapper*, como mostra a (Fig. 8), ele serve como unificador e controlador do *CacheClient* dentro do restante do código, possibilitando que o profissional passe, em sua instanciação, a dependência, que nesse caso é um agente gerenciador de cache, adaptado para a aplicação.

Por fim, na (Fig. 9), pode-se ver um caso de uso real, dentro do código, desses conceitos e padrões aplicados.

C. O domínio da aplicação

Na camada *Domain*, como supradito, é onde encontra-se o *core business* e todas as regras de negócios da aplicação, tudo o que é particular do sistema e compõe a sua singularidade. Na (Fig. 10), entende-se que as entidades e os casos de uso, nesse cenário, são os pilares da regra de negócio de todo o restante da aplicação. De acordo com a *Dependency Rule*, também supracitada, tudo deve convergir sua dependência para essa

camada, as entidades e os casos de uso ditam como cada regra deve funcionar e tudo em volta é baseado neles.

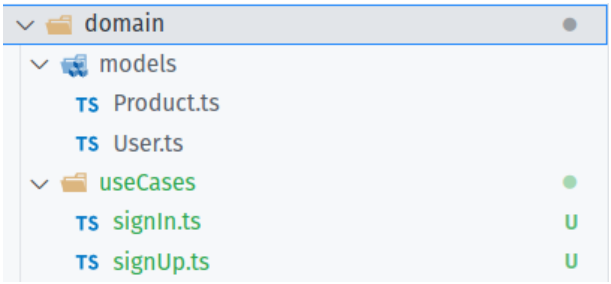


Fig. 10. Camada *domain*

Nesse sistema, as entidades existentes são *Product* e *User*, e os casos de uso são *signIn* e *signUp*, por si só entende-se quais são suas responsabilidades. Sua finalidade é ditar como o sistema deve reagir e prosseguir dentro do que é delimitado dentro deles, através do código.

D. A camada de infraestrutura

Como referenciado, essa camada é responsável por lidar com a implementação dos agentes externos e servir como plataforma e literalmente a infraestrutura de funcionamento da aplicação.

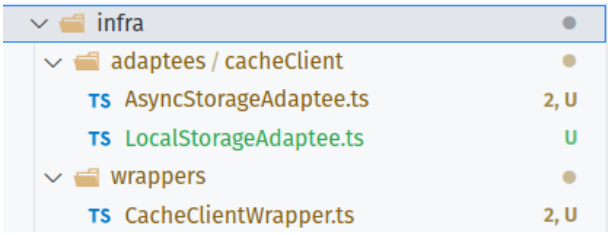


Fig. 11. Camada *infra*

Como é possível ver na (Fig 11), essa camada é responsável, nessa aplicação, por implementar os adaptadores do *CacheClientAdapter* e os respectivos *wrappers* que servem como ponto centralizado para uso dos agentes.

E. A camada main

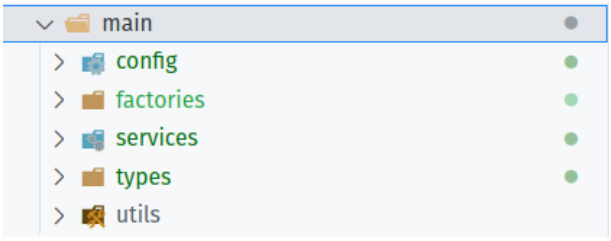


Fig. 12. Camada *main*

Na (Fig. 12), entende-se que essa camada é responsável por gerenciar exclusividades do projeto que não são, propriamente, o que compõe as regras de negócios. Exemplos disso são os *types*, que servem para tipagens gerais dentro do código e os *utils*, responsáveis por utilidades gerais dentro de todo o projeto.

F. A camada presentation

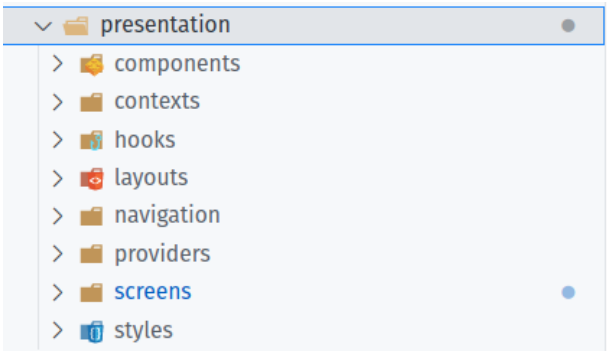


Fig. 13. Camada *presentation*

A camada representada pela (Fig. 13), é onde fica a parte do que deverá ser apresentado ao usuário final da aplicação. No caso dessa sistema, é onde fica o framework front-end, ele continua podendo trabalhar como sempre trabalhou, porém de forma desacoplada e dentro, somente, de suas responsabilidades.

De acordo com os fatos obtêm-se, através da segregação e desacoplamento dos componentes, um projeto onde as principais regras de negócios são agnósticas e podem ser modeladas livremente. Modelando de forma correta a aplicação dentro da arquitetura escolhida, é possível organizar e escalar a aplicação, de acordo com suas camadas e regras de fluxo.

VI. Conclusão

Neste trabalho, foi realizada a aplicação da *Clean Architecture* pensada no ecossistema do front-end. Através dos resultados obtidos e supramencionado, confirma-se que com a modelagem correta dessa estrutura, é possível obter uma aplicação segregada em suas responsabilidades, desacoplada, de fácil manutenibilidade, escalável, de alta disponibilidade e principalmente, com suas regras de negócios agnósticas em relação a qualquer outra camada ou *driver* externo, isso devido à sua *Dependency Rule*, igualmente supracitada.

Através desse método de trabalho, é possível garantir um código otimizado, organizado e testável, facilitando o ciclo de vida de qualquer manutenção ou desenvolvimento de novos recursos futuros. Ainda nesse ponto, outros profissionais da área precisarão de muito menos tempo para aprender a lidar com o projeto, pois é uma estrutura padronizada e conhecida pelo mercado.

Contudo, não existe verdade absoluta, a adoção dessa arquitetura deve ser avaliada perante ao projeto, em relação ao custo-benefício. Algumas vezes outras estruturas serão mais indicadas e se encaixarão melhor na solução do problema, em outras, essa estrutura se mostrará extremamente trabalhosa para algo muito pequeno.

REFERÊNCIAS

[1] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 1st ed. USA: Prentice Hall Press, 2017.

[2] Unyleya, "Arquitetura de software: Entenda por que ela É tão importante!" Feb 2020. [Online]. Available: https://blog.unyleya.edu.br/bitbyte/arquitetura-de-software

[3] D. Garlan, "Software architecture," Jun 2018. [Online]. Available: https://kilthub.cmu.edu/articles/journal_contribution/Software_Architecture/6609593/1

[4] "Robert cecil martin," Aug 2021. [Online]. Available: https://pt.wikipedia.org/wiki/Robert_Cecil_Martin

[5] J. Hunt, *Gang of Four Design Patterns*. Springer International Publishing, 2013. [Online]. Available: https://doi.org/10.1007/978-3-319-02192-8_16

[6] "Design patterns." [Online]. Available: https://refactoring.guru/pt-br/design-patterns/what-is-pattern

[7] J. E. McDonough, *Adapter Design Pattern*. Apress, 2017. [Online]. Available: https://doi.org/10.1007/978-1-4842-2838-8_15

[8] A. Harrer, N. Pinkwart, B. M. McLaren, and O. Scheuer, "The scalable adapter design pattern: Enabling interoperability between educational software tools," *IEEE Transactions on Learning Technologies*, vol. 1, no. 2, pp. 131–143, 2008.

[9] A. Cockburn, D. Haywood, Jonathan, B. K. O. Jacolyte, and F. A. Martins, "Hexagonal architecture," Oct 2021. [Online]. Available: https://alistair.cockburn.us/hexagonal-architecture/

[10] S. A. Smith, "Architecting modern web applications with asp.net core and microsoft azure," Nov 2021.

[11] M. E. Khalil, K. Ghani, and W. Khalil, "Onion architecture: a new approach for xaas (every-thing-as-a service) based virtual collaborations," in *2016 13th Learning and Technology Conference (L T)*, 2016, pp. 1–7.

[12] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 1997.

[13] "Barbara liskov," Nov 2021. [Online]. Available: https://pt.wikipedia.org/wiki/Barbara_Liskov

[14] I. Oktafiani and B. Hendradjaya, "Software metrics proposal for conformity checking of class diagram to solid design principles," in *2018 5th International Conference on Data and Software Engineering (ICoDSE)*, 2018, pp. 1–6.

[15] R. C. Martin, "Design principles and design patterns," *Object Mentor*, 2000.

[16] J. R. d. Paixão, "O que É solid: O guia completo para você entender os 5 princípios da poo," Jun 2020. [Online]. Available: 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

[17] H. Y. Yang, E. Tempero, and H. Melton, "An empirical study into use of dependency injection in java," in *19th Australian Conference on Software Engineering (aswec 2008)*, 2008, pp. 239–247.

[18] X. Chen, J. Chen, S. Zhang, and L. Sui, "Application of adapter pattern in container ship stowage system," in *2010 2nd International Conference on Industrial and Information Systems*, vol. 1, 2010, pp. 120–123.

[19] S. Rajam, R. Cortez, A. Vazhenin, and S. Bhalla, "Enterprise service bus dependency injection on mvc design patterns," in *TENCON 2010 - 2010 IEEE Region 10 Conference*, 2010, pp. 1015–1020.

[20] J. Deacon, "Model-view-controller (mvc) architecture," *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*, 2009.

[21] S. Boukhary and E. Colmenares, "A clean approach to flutter development through the flutter clean architecture package," in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2019, pp. 1115–1120.

[22] E. C. Arango and O. L. Loaiza, "Scrum framework extended with clean architecture practices for software maintainability," in *Software Engineering and Algorithms*, R. Silhavy, Ed. Cham: Springer International Publishing, 2021, pp. 667–681.

[23] A. Sondha, U. Saadah, F. Hardiansyah, and M. Rasyid, "Framework dan code generator pengembangan aplikasi android dengan menerapkan prinsip clean architecture," *Jurnal Nasional Teknik Elektro dan Teknologi Informasi*, vol. 9, pp. 327–335, 12 2020.

[24] S. Boukhary and E. Colmenares, "A clean approach to flutter development through the flutter clean architecture package," in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2019, pp. 1115–1120.

[25] D. Bui, "Reactive programming and clean architecture in android development," 2017.

[26] K. Sena Empreendedor, "Clean architecture - infra layer: Adapters," Apr 2021. [Online]. Available: https://www.yiiacademy.com.br/2021/04/11/5-clean-architecture-infra-layer-adapters/

[27] R. Wieruch, *The Road to Learn React: Your Journey to Master Plain Yet Pragmatic React. Js*. CreateSpace Independent Publishing Platform, 2017. [Online]. Available: https://books.google.com.br/books?id=NFD6swEACAAJ

[28] A. Boduch, *Flux architecture*. Packt Publishing Limited, 2016.