Utilização da Arquitetura Limpa para desenvolvimento de aplicativo mobile para gerenciamento de comandas *

Lucas Borghetti Araldi† Jorge Luis Boeira Bavaresco ‡

2022

Resumo

Este artigo apresenta a utilização da Arquitetura Limpa para o desenvolvimento de um aplicativo de comandas. Para isso o trabalho incorpora serviços de API em Node JS, armazenamento de dados no dispositivo e a utilização do framework Flutter. Além do desenvolvimento da aplicação baseada nos conceitos da Arquitetura Limpa, serão mostradas as melhorias e benefícios obtidos com sua utilização no desenvolvimento do aplicativo em relação ao antigo que será descontinuado.

Palavras-chaves: Flutter. Dart. Node Js. SQLite. Arquitetura Limpa.

1 Introdução

A modernização de bares e restaurantes tem sido cada vez mais atraente tendo em vista que passa mais confiança e um aspecto de modernidade para o estabelecimento. Levando em conta esses pontos a procura por tecnologias para empregar funções obsoletas como as de comandas tem sido cada vez mais frequente. A utilização de uma comanda digital traz mais agilidade e rapidez para o estabelecimento. Para o emprego de tal tecnologia é fundamental o uso de uma linguagem moderna e um código limpo para o desenvolvimento e a manutenibilidade da aplicação. A utilização de tablets com aplicativos mobile é a melhor forma de tornar prático para os estabelecimentos. O emprego de uma linguagem moderna como o Flutter da Google torna o aplicativo muito mais fluído principalmente em dispositivos Android e IOS. O padrão de design da Arquitetura Limpa segue o princípio da organização do código em camadas, assim ficando fácil a manutenção ou mesmo quando se

*Trabalho de Conclusão de Curso (TCC) apresentado ao Curso de Bacharelado em Ciência da Computação do Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense, Campus Passo Fundo, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação, na cidade de Passo Fundo, em 2022.

†<lucasaraldi.pf184@academico.ifsul.edu.br>

‡<jorgebavaresco@ifsul.edu.br>

torna obsoleto e precisa da substituição de uma das camadas se torne muito mais fácil. O objetivo deste trabalho é aplicar os conceitos desta arquitetura para obter um código limpo e desacoplado, permitindo um reaproveitamento de código e uma maior facilidade nas manutenções futuras, facilitando o trabalho em equipe, aonde cada membro da equipe pode trabalhar em uma camada, tornando-a independente, assim não necessitando de outras camadas prontas para a implementação da que está em desenvolvimento em questão.

O artigo está organizado da seguinte maneira: A seção 2 apresenta o referencial teórico, após a seção 3 detalha o desenvolvimento e por fim, a seção 4 contém as considerações finais apresentando sugestões de melhorias futuras.

2 REFERENCIAL TEÓRICO

Nesta seção serão apresentadas as tecnologias que foram utilizadas nesse projeto, o conceito que foi aplicado ao projeto e um trabalho relacionado.

2.1 TECNOLOGIAS UTILIZADAS

2.1.1 NodeJS

Node.js trata-se de uma tecnologia usada para executar código JavaScript fora do navegador. É um software de código aberto, multiplataforma baseado no interpretador V8 da Google. A principal característica dessa tecnologia em comparação com outras (PHP, Java e C#) é a execução das Requisições/Eventos em single-thread (chamada de Event Loop), onde apenas uma thread é responsável por executar o código JavaScript, sem a necessidade de criar uma nova thread que utiliza ainda mais recursos computacionais (LENON, 2018).

2.1.2 Flutter

O Flutter é o framework cross-platform criado pela empresa Google para o desenvolvimento de aplicativos, que foi lançado no final de 2018. A linguagem de programação utilizada pelo Flutter é o Dart desenvolvida pela Google, que foi criada em 2011 (FLUTTER, 2018).

O Flutter é baseado em widgets, como se o aplicativo fosse um lego e o widget fosse uma peça dele, assim no final do mesmo modo que várias peças compõem um brinquedo, vários widgets compõem uma aplicação. O Flutter ainda conta com o sistema de HotReload onde quando estamos com o projeto em depuração e fizemos alguma alteração a mesma já atualiza no projeto em execução, assim tornando muito mais rápido para visualizar as mudanças realizadas, como testá-las, uma opção interessante para mudanças gráficas (FLUTTER, 2018)

2.1.3 SQLite

O SQLite é uma base de dados relacional de código aberto e que dispensa o uso de um servidor na sua atuação. Armazenando seus arquivos dentro de sua própria estrutura. O SQLite suporta padrão dos bancos de dados relacionais como a sintaxe SQL, operações e instruções preparadas (SQLITE, 2000).

O SQLite suporta dados do tipo TEXT (similar a String em Java), INTEGER (semelhante a LONG em Java) e REAL (semelhante a Double em Java). Todos os outros

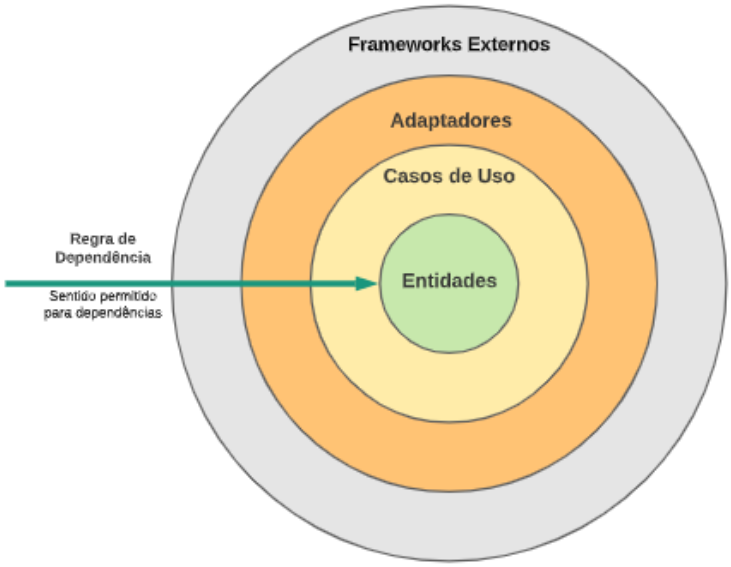
tipos devem ser convertidos pra algum desses tipos antes de armazená-los. O SQLite não valida os campos enviados para armazenamento, sendo dever do desenvolvedor validar os mesmos (SQLITE, 2000).

O SQLite é capaz de criar arquivos em disco, ler e escrever diretamente sobre este arquivo. O arquivo criado possui a extensão ".db". Uma tabela pode ser criada através do comando CRATE TABLE da linguagem SQL. Os dados das tabelas podem ser manipulados através de comandos DML (INSERT, UPDATE e DELETE) (SQLITE, 2000).

2.2 ARQUITETURA LIMPA

Segundo Martin (2020), este modelo de arquitetura visa ser usado no design de codificação a fim de facilitar a manutenção, testes e evolução de software através de um sistema de camadas conforme mostra a Figura 1, e um baixo grau de dependência.

Figura 1 – Camadas da Arquitetura Limpa



Fonte: Martin, 2020

Entidades: normalmente são responsáveis por regras de negócio que podem assumir entidades ou casos de usos. Como entidades podemos definir classes comuns para os vários sistemas da empresa, podemos também implementar regras de negócio seguindo de forma genérica.

Caso de uso: podem ser definidas como classes que são menos genéricas, usualmente relativas a um único sistema. Ao implementar um caso de uso podemos ver a regra de negócio que a define, exemplo seria um fluxo de cadastro com dados de uma pessoa física em uma empresa, o usuário terá como pré-requisito possuir um Cadastro de Pessoa Física – CPF para poder solicitar seu registro no sistema.

Adaptadores: possuem a função de converter ou adaptar dados de formatos para outro que possam ser entregues a camada de destino – seguindo o fluxo da regra de dependência.

Frameworks externos: nessa camada todas as implementações de classes perten-

cem a bibliotecas e frameworks externos ou de terceiros, eles podem definir vários recursos como persistência, construção de interfaces, solicitação de outros serviços como envio de e-mail, comunicação de diferentes hardwares com o software. Um ponto muito importante em relação às camadas definidas, quanto mais externas mantemos o fluxo de dependência sobre a camada inferior.

2.3 IOC - INVERSÃO DE CONTROLE

Inversão de Controle ou Inversion of Control - conhecido pela Sigla IoC é um Pattern que prega para usar o controle das instancias de uma determinada classe ser tratada externamente e não dentro da classe em questão, ou seja, Inverter o controle de uma classe delegando para uma outra classe, interface, componente, serviço (SOARES, 2018).

A utilização da inversão de controle no Flutter se faz através de um *plugin* disponível na pub.dev através do link: <https://pub.dev/documentation/ioc/latest/>.

2.4 TRABALHOS RELACIONADOS

Nesta seção são apresentados trabalhos que fazem o uso da mesma tecnologia escolhida e padrão arquitetural.

2.4.1 Desenvolvimento de um aplicativo utilizando o framework Flutter e Clean Architecture

A proposta do trabalho de conclusão de curso é um aplicativo mobile realizada por (BUENO, 2021), que realiza a autenticação de usuário através de e-mail e senha. A aplicação tem como padrão arquitetural a arquitetura limpa. O aplicativo tem camadas bem definidas, seguindo à risca o padrão arquitetural, sendo elas: Domain, Data, Infra, UI, Presentation e Validation, abaixo segue a definição de cada uma delas.

Domain (Domínio): São definidas as entidades que serão utilizadas pelos casos de uso e também são definidos contratos dos casos de uso, esta camada se relaciona diretamente com a camada de Entidades.

Data (dados): Esta camada é responsável por prover a implementação dos casos de usos assim como criar adaptadores que transformam dados em formatos externos para formatos compatíveis com as entidades. Aqui também serão feitas interfaces de contratos para os repositórios. Esta camada relaciona-se com a camada de Casos de Uso (Use Cases), ou seja, são as regras de negócio da aplicação e adaptadores de interface.

Infra: Esta camada é responsável por implementar os contratos de repositórios definidos na camada de dados. Aqui serão utilizadas bibliotecas externas como requisições web e persistência de dados. Esta camada relaciona-se com o último nível (Frameworks e Drivers).

UI (User Interface): Esta camada é responsável pela criação das interfaces de usuário e também da definição de contratos dos presenters e, relacionando-a com o modelo de Arquitetura Limpa, está pertence a camada de Frameworks e Drivers.

Presentation: A camada de Presentation é responsável por fazer a conexão entre a camada de dados (Data) e a interface do usuário (UI). Nesta camada dados são providos dos casos de uso para a interface e da interface para os casos de uso.

Validation: Esta camada tem como funcionalidade validar informações inseridos na camada de Presentation, como: E-mail, telefone, e etc.

A diferença entre o trabalho pesquisado e este desenvolvido, baseia-se a aplicação do aplicativo, enquanto o do autor em questão é uma simples tela de login e senha, o que está sendo desenvolvido não faz o uso de login e senha porém tem uma série de funcionalidades totalmente diferentes, voltado a realizar pedidos em bares e restaurantes. A aplicação deste projeto é uma migração de um aplicativo defasado para um com novas tecnologias, um padrão arquitetural assim tendo um upgrade.

2.4.2 A Clean Approach to Flutter Development through the Flutter Clean Architecture Package

A proposta do trabalho é uma abordagem limpa para programação em Flutter utilizando a Arquitetura Limpa por Boukhary e Colmenares (2019), o objetivo do mesmo é controlar o gerenciamento de estados no framework Flutter, fazendo o uso da Arquitetura Limpa foi possível obter esse objetivo proposto. O autor apresenta divisões de camadas sendo elas:

Domain: contem regras de negócios e entidades;

Device: repositórios e funções uteis ao projeto;

Data: repositórios, constantes como rotas de api e componentes como o http;

App: Estados de Widget, Navegações e manipuladores de estados.

O autor implementou testes unitários na aplicação para verificar o comportamento das menores unidades em sua aplicação. Com a utilização da Arquitetura Limpa foi possível delegar desenvolvimento para diferentes programadores que estavam no projeto, devido a separação das camadas, a rápida detecção de *"bugs"* também foi um beneficio obtido, visto que os testes foram "quebrados" em camadas.

A fim de testar a independência da arquitetura, o autor fez a troca de banco de dados de SQL para MongoDB e a aplicação continuou funcionando sem nenhuma outra alteração além da camada aonde se encontra o banco de dados.

O trabalho proposto por Boukhary e Colmenares (2019), propõem a solucionar o problema de gerenciamento de estado do Flutter através da Arquitetura Limpa, o mesmo faz a separação de camadas como este desenvolvido, porém o autor implementa testes unitários otimizando a correção de bugs por estarem separados em camadas. A troca de banco de dados veio a validar que a arquitetura proposta de modo com que não seja alterada nada além da camada aonde se encontra o banco de dados. O trabalho pesquisado veio a auxiliar na separação de camadas do que foi desenvolvido.

3 APLICAÇÃO DA ARQUITETURA LIMPA

Nesta seção é apresentado como foi realizado o desenvolvimento da aplicação e a introdução da Arquitetura Limpa no mesmo. Serão abordados casos de uso da aplicação, pontos da arquitetura e componentes que compõem o projeto.

Visando o objetivo do trabalho de obter um código limpo e desacoplado com fácil manutenibilidade, será explicado nas próximas seções aspectos desse desenvolvimento.

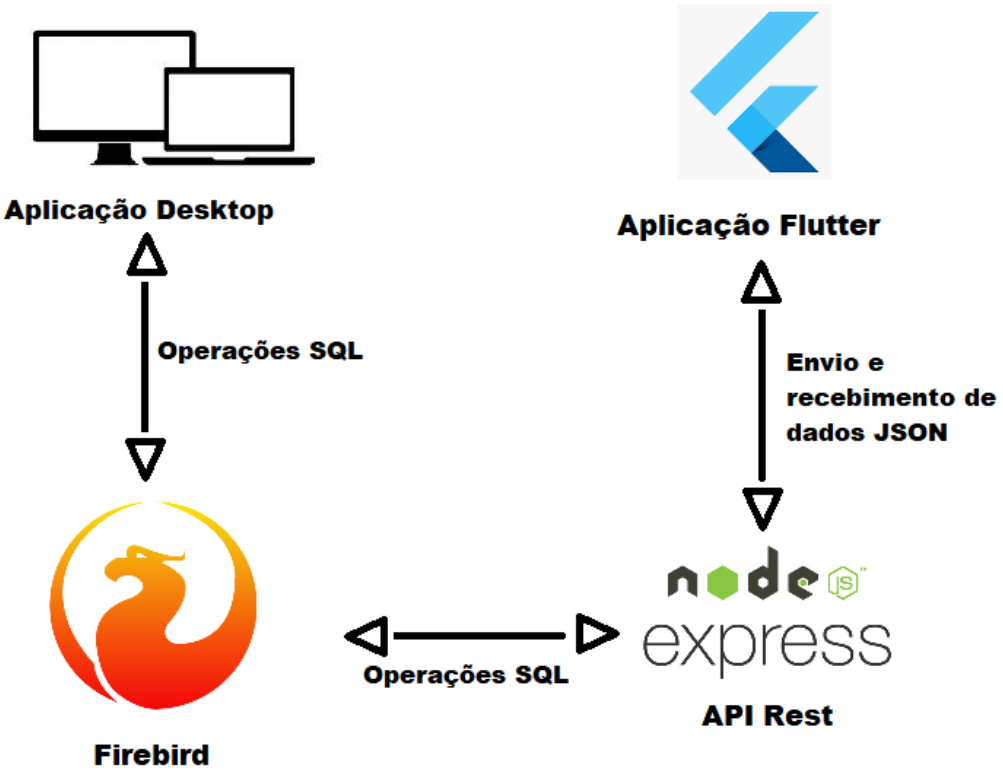
3.1 VISÃO GERAL DOS COMPONENTES DA SOLUÇÃO

A solução para o estudo conta com alguns componentes para o seu funcionamento em questão, sendo eles: aplicação em Flutter, API em Node Js, banco de dados Firebird e uma aplicação Desktop de uma empresa parceira.

A aplicação em Flutter emprega os padrões da Arquitetura Limpa assim, sendo o principal componente desse estudo. A API em Node Js irá realizar a comunicação entre a aplicação em Flutter e o banco de dados Firebird, inserindo e buscando dados. O banco de dados Firebird contém os dados do sistema Desktop de terceiros onde será feita uma busca dos produtos para compor uma comanda e posteriormente gravar a mesma. A aplicação desktop de uma empresa parceira, fará o gerenciamento das comandas, assim disponibilizando para a empresa várias outras ferramentas, como emissão de nota fiscal, controle de estoque, impressões e relatórios dos movimentos.

A Figura 2 demonstra esse fluxo de informações entre os componentes utilizados no estudo, desde a aplicação em Flutter até a aplicação desktop.

Figura 2 – Diagrama representando os componentes de forma geral



Fonte: Autor, 2022

3.2 CASOS DE USO

Para demonstrar a aplicação da arquitetura, nesse projeto foram separados dois casos de usos que contemplam todas as camadas da arquitetura limpa, são eles: a página de configuração e o atualizar dados.

A página de configuração armazena dados, como: endereço de IP do servidor, código da empresa no ERP, que consiste no código identificador do cliente na aplicação desktop e

modo de operação do aplicativo, sendo eles, bar, restaurante e pizzaria. Os mesmos serão armazenados de forma local no dispositivo.

O segundo caso de uso da aplicação é o "atualizar dados", onde externamente são buscados os dados referente aos produtos que vão compor a comanda futuramente. Assim com estes dois casos de uso consegue-se contemplar todas as camadas da Arquitetura Limpa proposta para esta aplicação.

3.3 VISÃO GERAL DA ARQUITETURA

A aplicação foi dividida em camadas sendo elas, *domain*, *infrastructure* e *view*

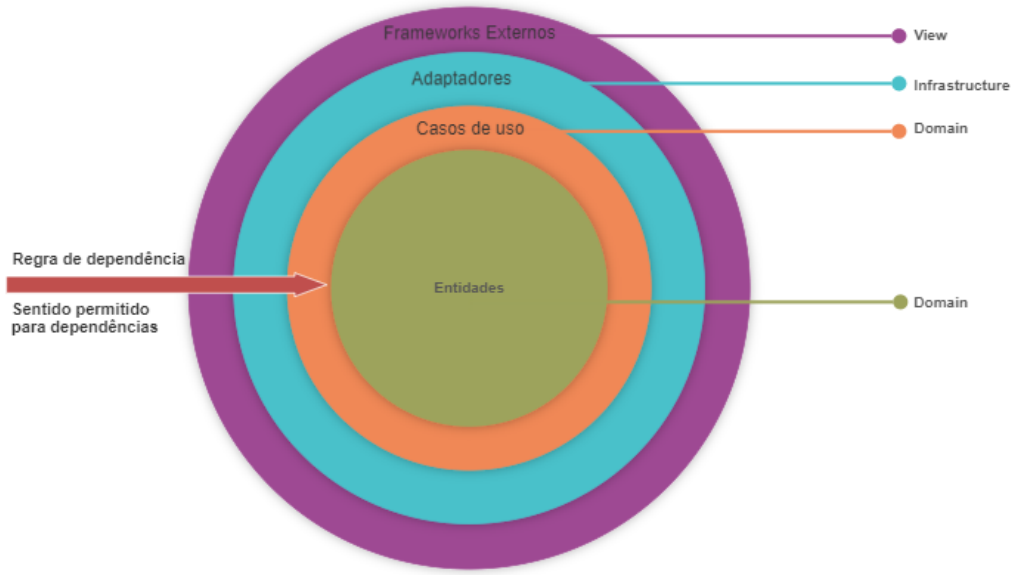
Domain (Domínio): Nesta camada são definidas as entidades que serão utilizadas pelos casos de uso e também são definidos os contratos que serão implementados na camada de infrastructure;

Infrastructure: Esta camada é responsável por implementar os contratos de repositórios definidos na camada Domain. Aqui serão utilizadas bibliotecas externas como requisições web e persistência de dados;

View: Aqui é onde realmente o framework Flutter será utilizado pois esta camada é responsável pela criação da interface do usuário. Esta camada relaciona-se diretamente com a camada de Framework e Drivers em Arquitetura Limpa.

Como pode-se perceber na Figura 3, a camada mais interna contém a camada "entidades", e na camada acima dela constam os "casos de uso", formando a camada entidades do nosso projeto, acima dela adaptadores que é a camada "infrastructure" do nosso projeto e "frameworks externos" que é a camada de "view". No lado esquerdo pode-se ver uma seta apontando para o centro da figura, mais especificamente para as entidades, essa seta indica a regra de dependência que a arquitetura possui, assim apenas as camadas de fora tem conhecimento das camadas mais internas, onde como por exemplo as entidades não podem depender dos frameworks externos para funcionar, mas os mesmos podem conter entidades para o seu funcionamento.

Figura 3 – Figura demonstrando a regra e junto as camadas da aplicação com as regas de dependência



Fonte: Autor, 2022

3.4 AMBIENTE DE DESENVOLVIMENTO

Para o desenvolvimento da aplicação foram utilizadas algumas ferramentas, sendo elas:

- **Visual Studio Code:** versão 1.69.1;
- **Dart:** versão 2.17.5;
- **Flutter:** versão 3.0.4;
- **SQLite:** versão 2.0.3;
- **Node Js:** versão 16.13.2;
- **Emulador do Android:** versão 11.0.

3.5 ESTRUTURA DE PASTAS DO PROJETO

A estrutura de pastas deve ser definida de modo que possa ser identificado rapidamente onde cada funcionalidade deve estar. A Figura 4 ilustra uma visão da estruturação proposta para esse trabalho. estas pastas demonstram as camadas da aplicação, sendo elas:

- Domain;
- Infrastructure;
- view.

Figura 4 – Estrutura Arquitetura Limpa no Flutter

```
app/
 domain/                          <--- camada de dominio
  entities/                        <--- entidades
   cardapio.dart                    <--- entidade cardapio
   comanda.dart                     <--- entidade comanda
   configuracao.dart                <--- entidade configuracao
   finalizacao.dart                 <--- entidade finalizacao
   grupo.dart                       <--- entidade grupo
   produto.dart                     <--- entidade produto
   segmento.dart                    <--- entidade segmento
  interfaces/                      <--- metodos abstratos que definem as funcionalidades
   baseComandaDao.dart              <--- define as funcionalidades da ComandaDao
   baseConfiguracoesDao.dart        <--- define as funcionalidades da ConfiguracoesDao
   baseGrupoDao.dart                <--- define as funcionalidades da GrupoDao
   baseGrupoService.dart            <--- define as funcionalidades da GrupoService
   baseProdutoDao.dart              <--- define as funcionalidades da ProdutoDao
   baseProdutoService.dart          <--- define as funcionalidades da ProdutoService
   baseSegmentoDao.dart             <--- define as funcionalidades da SegmentoDao
   baseSegmentoService.dart         <--- define as funcionalidades da SegmentoService
  services/                        <--- classes com servicos de atualizações de dados
   syncData.dart                    <--- invoca outros componentes externos para busca de dados
 infrastructure/                  <--- banco de dados e serviços de busca externa
  databases/                       <--- camada que contem todas as base de dados
   sqlite/                          <--- base dados sqlite
    dao/                             <--- dao que contem as operações de banco de dados
     comandasDao.dart                 <--- classe que implementa baseComandaDao
     configuracoesDao.dart            <--- classe que implementa baseConfiguracoesDao
     finalizacoesDao.dart             <--- classe que implementa baseFinalizacoesDao
     gruposDao.dart                   <--- classe que implementa baseGrupoDao
     produtosDao.dart                 <--- classe que implementa baseProdutoDao
     segmentosDao.dart                <--- classe que implementa baseSegmentoDao
    appDatabase.dart                 <--- classe que cria e define o banco de dados
  services/                        <--- classes que realizam a busca externa de dados
   segmentoService.dart             <--- envia para api objetos do tipo comanda
   grupoService.dart                <--- busca os grupos de produtos
   produtoService.dart              <--- busca os produtos
   segmentoService.dart             <--- busca os segmentos
 view/                            <-- camada view (paginas, wigtes e outro componentes)
  configuration.dart               <--- pagina de configuração
  grupoList.dart                   <--- pagina de listagem de grupos de proutos
  homePage.dart                    <--- pagina inicial
  produtoList.dart                 <--- pagina de listagem de produtos
 myApp.dart                       <--- rotas de navegação e inicia a pagina inicial
main.dart                        <--- classe inicial da aplicação
```

Fonte: Autor, 2022

3.6 CAMADA DE DOMÍNIO

Na camada de domínio da aplicação são definidas as entidades, interfaces e services.

Na Figura 5 pode-se visualizar as entidades, essas são classes comuns a aplicação, que não dependem de nenhuma outra, contendo os atributos que as definem, como pode-se ver na Figura 5, onde os atributos “cod_empresa”, “modo_operacao” e “ip” pertencem a entidade “Configuracao”.

Nas interfaces definimos os métodos abstratos, como pode-se ver na Figura 6 que serão implementados posteriormente em classes que utilizarão os métodos. As interfaces contém “contratos” onde detalham como a classe que a implementa deve definir os seus métodos. É mostrada a interface chamada “baseConfiguracaoDao” que posteriormente vai ser implementada pelo método Dao respectivo a essa classe.

Figura 5 – Entidade configuracao

lib > app > domain > entities > configuracao.dart > Configuracao

```dart
class Configuracao {
  final int cod_empresa;
  final String modo_operacao;
  final String ip;
  int checked;

  Configuracao(this.cod_empresa, this.modo_operacao, this.ip,
      [this.checked = 0]);

  @override
  String toString() {
    return 'Configuracao{Cod empresa: $cod_empresa, Modo operação: $modo_operacao, IP: $ip}';
  }
}

```

Fonte: Autor, 2022

Figura 6 – Interface baseConfiguracoesDao

lib > app > domain > interfaces > baseConfiguracoesDao.dart > ...

```dart
import '../entities/configuracao.dart';

abstract class baseConfiguracoesDao {
  Future<int> save(Configuracao configuracao);
  Future<int> update(Configuracao configuracao);
  Future<int> delete(int id);
  Map<String, dynamic> toMap(Configuracao configuracao);
  Future<List<Configuracao>> findAll();
  List<Configuracao> toList(List<Map<String, dynamic>> result);
  Future<List<Map<String, dynamic>>> buscarTodos();
}
```

Fonte: Autor, 2022

Os “services” são métodos onde são recebidos os objetos para serem armazenados no banco de dados local, como pode ser visto na Figura 7, no seu primeiro método, chamado “atualizaGrupo”, é possível ver que a partir da inversão de controle é buscada a instância da classe “baseGrupoService”. Obtém-se dados no formato Json provenientes de uma API externa, outra instancia é recuperada, a “baseGrupoDao”, a partir de um método definido dentro da entidade grupo, consegue-se converter os dados externos e gravar os mesmos no banco de dados local.

Figura 7 – Service syncData

Fonte: Autor, 2022

3.7 CAMADA DE INFRASTRUCTURE

A camada de infrastructure consiste em tudo o que existe independentemente da aplicação, bibliotecas externas, mecanismo de banco de dados e mecanismos de busca externa.

Na Figura 8 pode-se ver a classe que recupera o banco de dados local, a partir dela também o próprio componente verifica se as tabelas já estão criadas, caso o contrário já cria seguindo os padrões estipulados no Dao.

As classes DAO implementam as interfaces declaradas nas “interfaces” da camada “domain”, como pode ser visualizado na Figura 9. Nela pode-se visualizar a implementação dos métodos como save, update e delete e os atributos que irão gerar o nome das tabelas no banco de dados

Os services realizam a busca de dados externos como é ilustrado na Figura 10, os “services” contém as rotas da API e componentes para realizar a busca nas mesmas.

Figura 8 – appDatabase.dart

lib > app > infrastructure > database > sqlite > appDatabase.dart > ...

```dart
import 'dart:async';
import 'package:path/path.dart';
import 'package:sqflite/sqflite.dart';
import '../sqlite/dao/comandasDao.dart';
import '../sqlite/dao/segmentosDao.dart';
import '../sqlite/dao/gruposDao.dart';
import '../sqlite/dao/produtosDao.dart';
import '../sqlite/dao/finalizacoesDao.dart';
import '../sqlite/dao/configuracoesDao.dart';

Future<Database> getDatabase() async {
  final String path = join(await getDatabasesPath(), 'dbcomandas.db');
  return openDatabase(
    path,
    onCreate: (db, version) {
      db.execute(comandasDao.tableSQL);
      db.execute(segmentosDao.tableSQL);
      db.execute(gruposDao.tableSQL);
      db.execute(produtosDao.tableSQL);
      db.execute(finalizacoesDao.tableSQL);
      db.execute(configuracoesDao.tableSQL);
    },
    version: 1,
    onDowngrade: onDatabaseDowngradeDelete,
  );
}
```

Fonte: Autor, 2022

Figura 9 – configuracoesDao.dart

lib > app > infrastructure > database > sqlite > dao > configuracoesDao.dart > ...

```dart
import 'package:app_comandas/app/domain/entities/configuracao.dart';
import 'package:app_comandas/app/domain/interfaces/baseConfiguracoesDao.dart'
import 'package:sqflite/sqflite.dart';
import '../appDatabase.dart';

class configuracoesDao implements baseConfiguracoesDao {
  static const String tableSQL = 'CREATE TABLE $_tableName('
      '$_cod_empresa INTEGER PRIMARY KEY, '
      '$_modo_operacao VARCHAR(100), '
      '$_ip VARCHAR(100)); ';

  static const String _tableName = 'configuracao';
  static const String _cod_empresa = 'cod_empresa';
  static const String _modo_operacao = 'modo_operacao';
  static const String _ip = 'ip';

  Future<int> save(Configuracao configuracao) async {
    final Database db = await getDatabase();
    Map<String, dynamic> configuracaoMap = toMap(configuracao);
    return db.insert(_tableName, configuracaoMap);
  }

  Future<int> update(Configuracao configuracao) async {
    final Database db = await getDatabase();
    Map<String, dynamic> configuracaoMap = toMap(configuracao);
    return db.update(_tableName, configuracaoMap);
  }

  Future<int> delete(int id) async {
    final Database db = await getDatabase();
    return db.delete(_tableName, where: '$_cod_empresa: ?', whereArgs: [id]);
  }

  Map<String, dynamic> toMap(Configuracao configuracao) {
    final Map<String, dynamic> configuracaoMap = Map();
    configuracaoMap[_cod_empresa] = configuracao.cod_empresa;
    configuracaoMap[_modo_operacao] = configuracao.modo_operacao;
    configuracaoMap[_ip] = configuracao.ip;
    return configuracaoMap;
```

Fonte: Autor, 2022

Figura 10 – produtoService.dart

lib > app > infrastructure > services > produtoService.dart > ...

```dart
import 'package:dio/dio.dart';
import 'package:ioc/ioc.dart';

import '../../domain/interfaces/baseConfiguracoesDao.dart';
import '../../domain/interfaces/baseProdutoService.dart';

// Response response;
class produtoService implements baseProdutoService {
  Future<Response<dynamic>> getProdutos() async {
    final db = Ioc().use<baseConfiguracoesDao>('baseConfiguracoesDao');
    List lista = await db.buscarTodos();
    var config = lista[0];
    String _ip = config['ip'];
    String _url = 'http://' + _ip + ':3000/produtos';

    try {
      var dio = Dio();
      var response = await dio.get('$_url',
          options: Options(
            responseType: ResponseType.plain,
          ));
      return response;
    } catch (e) {
      print(e);
    }
    return Future.value(null);
  }
}
```

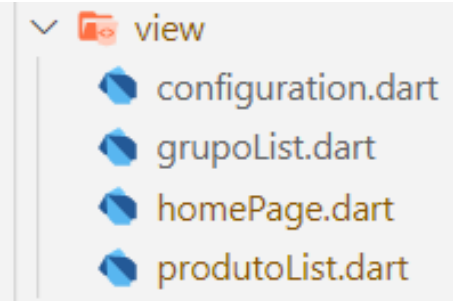
Fonte: Autor, 2022

3.8 CAMADA VIEW

A camada view é a mais externa, onde se implementa tudo que foi criado anteriormente, nela ficam as páginas e widgets do Flutter, onde foram criadas as telas, partes visuais da aplicação. A Figura 11 apresenta as classes que compõem as páginas da aplicação.

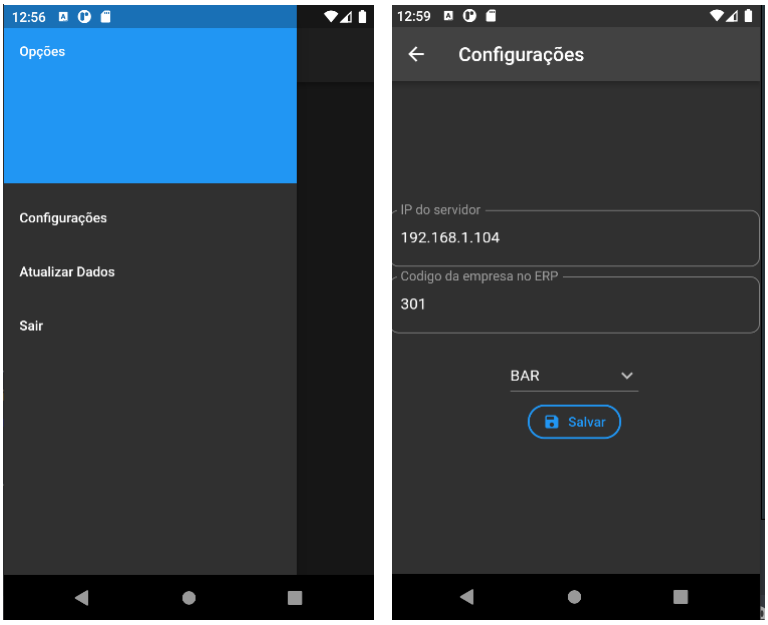
Na Figura 12(a) pode-se visualizar a opção atualizar dados que representa o caso de uso atualizar dados onde busca-se externamente dados para gravar no banco de dados local do dispositivo. Na Figura 12(b) pode-se visualizar a página de configuração onde representa o caso de uso que gerencia algumas configurações que determinam o funcionamento da aplicação. Nessa tela define-se o IP do servidor onde será realizado a busca dos dados, o código da empresa no ERP e também o modo de operação bar, pizzaria ou restaurante.

Figura 11 – Camada view



Fonte: Autor, 2022

Figura 12 – Telas da aplicação



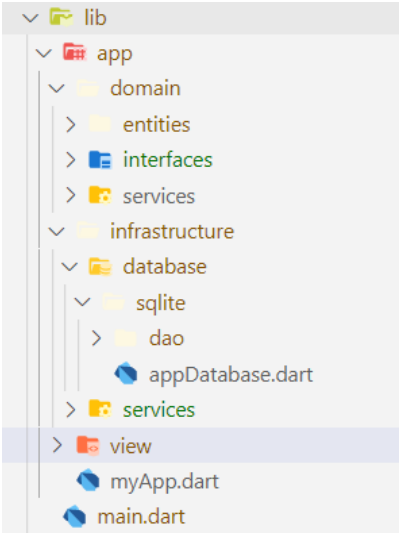
(a) Menus de escolhas (b) Telas de configuração

Fonte: do autor, 2022

3.9 JUNÇÃO DAS CAMADAS

Com o emprego da Arquitetura Limpa na aplicação, é obtido um código desacoplado e de fácil manutenibilidade, no final é visualizada uma estrutura de pastas maior que a de uma aplicação sem os mesmos conceitos. Na Figura 13 é possível visualizar as divisões de pastas e camadas.

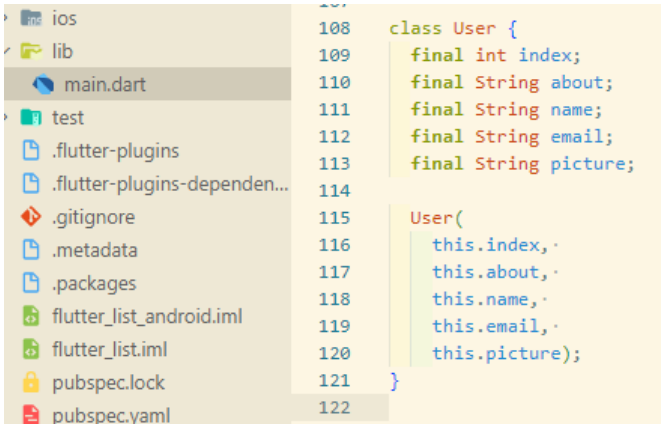
Figura 13 – Estrutura de pastas do projeto



Fonte: Autor, 2022

As Figura 14(a) e Figura 14(b) demonstram como seria uma aplicação sem os padrões da Arquitetura Limpa, uma única classe chamada “main”, sem estruturação de pastas, sem padrões a seguir e sem reaproveitamento de código. A Figura 14(a) mostra como foi declarada a entidade dentro do “main” da aplicação e a Figura 14(b) demonstra como buscar dados, também dentro da mesma classe, assim conclui-se que essa classe principal do projeto está “suja”, pois não segue nenhum padrão de arquitetura, tornando o código acoplado, dependente e de difícil manutenção.

Figura 14 – Exemplo de uso de não utilização da arquitetura limpa



(a) Exemplo de entidade sem utilização da Arquitetura Limpa

(b) Exemplo de busca de dados externos sem utilização da Arquitetura

Fonte: do autor, 2022

Neste projeto foi utilizado a inversão de controle para interagir com as entidades de banco de dados SQLite e os serviços que se comunicam com a API externa. A Figura 15 apresenta o container declarado no main da aplicação, onde estão as instâncias dos widgets que serão usados posteriormente. A Figura 16 demonstra a utilização do IOC, onde a variável “db” recebe uma instância da classe que esta configurada no IOC.

Figura 15 – Configuração do Container no main

```
18  void main() {
19    configureContainer();
20    runApp(MyApp());
21  }
22
23  configureContainer() {
24    Ioc().bind('baseComandaDao', (ioc) => comandasDao());
25    Ioc().bind('baseGrupoDao', (ioc) => gruposDao());
26    Ioc().bind('baseGrupoService', (ioc) => grupoService());
27
28    Ioc().bind('baseProdutoDao', (ioc) => produtosDao());
29    Ioc().bind('baseProdutoService', (ioc) => produtoService());
30
31    Ioc().bind('baseComandaService', (ioc) => comandaService());
32
33    Ioc().bind('baseFinalizacaoDao', (ioc) => finalizacoesDao());
34
35    Ioc().bind('baseSegmentoDao', (ioc) => segmentosDao());
36    Ioc().bind('baseSegmentoService', (ioc) => SegmentoService());
37
38    Ioc().bind('baseConfiguracoesDao', (ioc) => configuracoesDao());
39  }
```

Fonte: Autor, 2022

Figura 16 – utilizando o IOC

```
15  class _ConfigurationState extends State<Configuration> {
16    final db = Ioc().use<baseConfiguracoesDao>('baseConfiguracoesDao');
17    List<Map<String, dynamic>> _configs = [];
18    bool _isLoading = true;
19    void _atualizarConfigs() async {
20      final data = await db.buscarTodos();
```

Fonte: Autor, 2022

4 CONSIDERAÇÕES FINAIS

O trabalho teve como objeto realizar um estudo da viabilidade de aplicar conceitos da Arquitetura Limpa para o desenvolvimento de um projeto no framework Flutter. O foco da Arquitetura Limpa é elaborar sistemas com camadas bem definidas, coesas e desacopladas. Por meio desta arquitetura foi possível obter uma aplicação com camadas bem definidas onde o usuário busca dados de uma aplicação de externa, cria uma comanda, adiciona os respectivos produtos e envia a mesma para uma API.

Para o desenvolvimento da aplicação, a documentação do framework Flutter forneceu todos os dados necessários para realizar a implementação da arquitetura. O ambiente de desenvolvimento supriu todas as necessidades para a realização do projeto.

O estudo e entendimento das regras da arquitetura foram de grande importância para a tomada de decisões durante o desenvolvimento. A aplicação da Arquitetura Limpa demonstrou-se viável no desenvolvimento de aplicações utilizando o framework Flutter, assim, bastando respeitar os princípios que a arquitetura propõe. Pode-se concluir que utilizando a Arquitetura Limpa são obtidos alguns benefícios como, código limpo e facilidade na manutenção. Por exemplo, quando tem a necessidade de trocar o banco de dados, simplesmente altera-se o arquivo Dao de configuração, seguindo os “contratos” que o

definem e a aplicação continua funcionando sem precisar se preocupar com as outras camadas. Outro benefício obtido foi o reaproveitamento de código, quando é seguido um padrão, as entidades são classes comuns a todas as camadas da aplicação, não sendo necessário recriar elas em outro momento, evitando a duplicidade de código.

Em contrapartida o tempo de desenvolvimento aumenta, tendo em vista que é preciso definir bem toda a estrutura antes mesmo de começar a codificar, é um tempo que deve-se considerar, visando os benefícios futuros como a manutenibilidade.

Como desenvolvimento futuro em vista do curto período de tempo para realização desse estudo, propõem-se a implementação de um gerenciador de estados, testes unitários e a injeção de dependências assim tornando o código totalmente desacoplado e completo.

USE OF CLEAN ARCHITECTURE IN MOBILE APPLICATION DEVELOPMENT FOR ORDER SHEET MANAGING

Lucas Borghetti Araldi* Jorge Luis Boeira Bavaresco †

2022

Abstract

This article presents the use of Clean Architecture for the development of a command application. For this, the work incorporates API services in Node JS, data storage on the device and the use of the Flutter framework. In addition development of the application based on the concepts of Clean Architecture, will be The improvements and benefits obtained with its use in the development of the application over the old one that will be discontinued.

Key-words: Flutter. Dart. Node Js. SQLite. Clean Architecture.

Referências

BOUKHARY, S.; COLMENARES, E. A clean approach to flutter development through the flutter clean architecture package. In: *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. [S.l.: s.n.], 2019. p. 1115–1120. Citado na página 5.

BUENO, C. E. de O. *Desenvolvimento de um aplicativo utilizando o framework Flutter e Arquitetura Limpa*. 47 f. Monografia (Graduação) — Faculdade de Educação, Pontifícia Universidade Católica de Goiás, Goias, 2021. Citado na página 4.

FLUTTER. *Flutter*. 2018. Disponível em: <https://docs.flutter.dev/>. Acesso em: 22 fev 2022. Citado na página 2.

*<lucasaraldi.pf184@academico.ifsul.edu.br>
†<jorgebavaresco@ifsul.edu.br>

LENON. *Introdução ao Node.js*. 2018. Disponível em: <https://www.opus-software.com.br/node-js/>. Acesso em: 10 jan 2022. Citado na página 2.

MARTIN, R. C. *Arquitetura Limpa: O guia do artesão para estrutura e design de software*. [S.l.]: Rio de Janeiro, Alta Books, 2020. Citado na página 3.

SOARES, M. P. L. *Inversão de Controle (IoC) e Injeção de Dependência (DI) - Diferenças*. 2018. Disponível em: <http://www.linhadecodigo.com.br/artigo/3418/inversao-de-controle-ioc-e-injecao-de-dependencia-di-diferencas.aspx>. Acesso em: 22 jul 2022. Citado na página 4.

SQLITE. *SQLITE*. 2000. Disponível em: <https://sqlite.org/index.html>. Acesso em: 22 fev 2022. Citado 2 vezes nas páginas 2 e 3.