

Desenvolvimento de uma Plataforma para a Gestão do Condomínio Madre Paulina

William S. Nepomuceno¹, Victor B. da Silva¹

¹ Instituto de Tecnologia – Universidade de Passo Fundo (UPF)
Passo Fundo – RS – Brazil

178344@upf.br, victorbilly@upf.br

Abstract. *This paper presents the proposal of a platform for the management of the Madre Paulina condominium and also the development of improvements to the existing WEB platform. Initially it will describe the requirements and use cases raised with the end users, and it will also present the final result of the development, with images of the screens and explanation of the functioning. Finally, there will be an explanation of the technical implementation of the development, going through the Clean Architecture concepts and their layers, the SOLID Principles and the results obtained by applying the test-driven development.*

Resumo. *O presente trabalho apresenta a proposta de uma plataforma para a gestão do condomínio Madre Paulina e também o desenvolvimento de melhorias para a plataforma WEB já existente. Inicialmente será descrito os requisitos e casos de usos levantados com os usuários finais, passando pela apresentação das funcionalidades desenvolvidas para o aplicativo e a plataforma WEB. Por fim, terá uma explicação sobre a implementação técnica do desenvolvimento, passando pelos conceitos Clean Architecture e suas camadas, os Princípios SOLID e os resultados obtidos ao aplicar o desenvolvimento orientado a testes.*

1. Introdução e Contextualização

Este projeto tem como objetivo o desenvolvimento de uma plataforma para a gestão do condomínio Madre Paulina, localizado em Tapejara - Rio Grande do Sul.

Atualmente o condomínio Madre Paulina conta com quinze condôminos, sendo duas salas comerciais, devido a quantidade de condôminos, para facilitar a gestão das tarefas diárias, o condomínio possui uma plataforma WEB para o gerenciamento de suas atividades, desenvolvida pelo próprio autor durante a disciplina de Laboratório de Engenharia de Software em 2022/1.

O backend da plataforma WEB foi desenvolvido utilizando o framework Laravel [9] e o frontend utilizando o framework Angular [2], que se comunicam através de uma API Rest. Os serviços de Backend e Frontend estão hospedados no Heroku [8], que é uma plataforma de hospedagem de serviços web. Para o banco de dados foi utilizado o SGBD MySQL [1], atualmente hospedado nos serviços RDS da Amazon Web Services [10].

O aplicativo foi desenvolvido utilizando a linguagem de programação Dart [4] e o framework Flutter [5], que permite o desenvolvimento de aplicações móveis para os sistemas operacionais Android e iOS, o aplicativo também irá se comunicar com o backend através de uma API Rest. O aplicativo está disponível para download na Play Store [7].

1.1. Motivação

A partir do cenário atual do condomínio Madre Paulina, aliado com o crescimento da demanda contábil sobre a administração de condomínios onde é cada vez mais necessário obter informações precisas para a tomada de decisão, a fim de evitar um déficit nas contas, e levando em consideração que neste condomínio ainda são realizados de forma manual em cadernos, causando transtornos na prestação de contas, pela perda de informações, urge a necessidade de uma ferramenta que possibilite e facilite a contabilidade do condomínio Madre Paulina de forma mais precisa.

1.2. Limitação

Devido ao alto custo para a publicação de uma versão do aplicativo para o sistema operacional iOS, a publicação do aplicativo estará limitada somente à plataforma Android. Devido ao curto prazo para o desenvolvimento, não foram realizadas avaliações de experiência com os usuários referentes a usabilidade e possíveis melhorias para a plataforma WEB e do aplicativo móvel.

2. Modelagem da aplicação

Na base da modelagem dos projetos estão os seus requisitos, é fundamental que ao se desenvolver um sistema saibamos quais as necessidades das partes interessadas: usuários, clientes, fornecedores, desenvolvedores, empresas e o que o sistema deverá fazer para satisfazer essas necessidades.

Também é importante ponderar que os requisitos da aplicação disponibilizaram a base para o planejamento do desenvolvimento aplicação, bem como critérios para testes.

2.1. Requisitos do sistema

Requisitos definem o que um sistema deve fazer e sob quais restrições. Requisitos relacionados com a primeira parte dessa definição — o que um sistema deve fazer, ou seja, suas funcionalidades, ou até mesmo o que um sistema não deve fazer — são chamados de Requisitos Funcionais. Já os requisitos relacionados com a segunda parte — sob que restrições — são chamados de Requisitos Não-Funcionais [11, Capítulo 3.1].

Após o levantamento dos requisitos com o usuário final da aplicação, foram identificados os seguintes Requisitos Funcionais [2.1.1] e Requisitos Não-Funcionais [2.1.2].

2.1.1. Requisitos Funcionais

Nesta seção serão descritos os dois principais requisitos funcionais solicitados pelo usuário final.

RF-001	Adicionar usuário
RF-002	Editar usuário
RF-003	Adicionar período de Receitas / Despesas
RF-004	Adicionar receita/despesa em um período
RF-005	Remover receita/despesa de um período
RF-006	Relatório do Fluxo de Caixa
RF-007	Cadastrar Condomínio
RF-008	Cadastrar Condômino
RF-009	Informar leitura d'água

Inicialmente estava previsto também o desenvolvimento de outras duas funcionalidades, sendo a Reversa do Salão de Festas e Atas de Assembleia, porém os usuários finais não chegaram em um consenso quanto as regras de negócios.

Outras funcionalidades, como a Integração com o Internet Baking para a geração dos boletos do condomínio, não foram desenvolvidas por falta de tempo.

2.1.2. Requisitos Não-Funcionais

Nesta seção serão descritos os dois principais requisitos não-funcionais.

Segurança	
RNFSEG-001	Apenas usuários cadastrados deverão ter acesso ao sistema, por meio de login.
RNFSEG-002	As requisições às apis deverão ser autenticadas por meio de Json Web Token.
RNFSEG-003	As senhas dos usuários deverão estar criptografadas.
Interface	
RNFINT-001	O aplicativo deve ter uma interface visual de fácil compreensão.

2.2. Casos de usos

Casos de uso são documentos textuais de especificação de requisitos. Um caso de uso enumera os passos que um ator (usuário) realiza em um sistema com um determinado objetivo. Um caso de uso inclui duas listas de passos, sendo a primeira o fluxo normal, que são os passos necessários para concluir uma operação com sucesso. A segunda representa extensões desse fluxo normal, geralmente casos de exceções [11, Capítulo 3.1]. A Figura 1 apresenta o Diagrama de Casos de Usos.

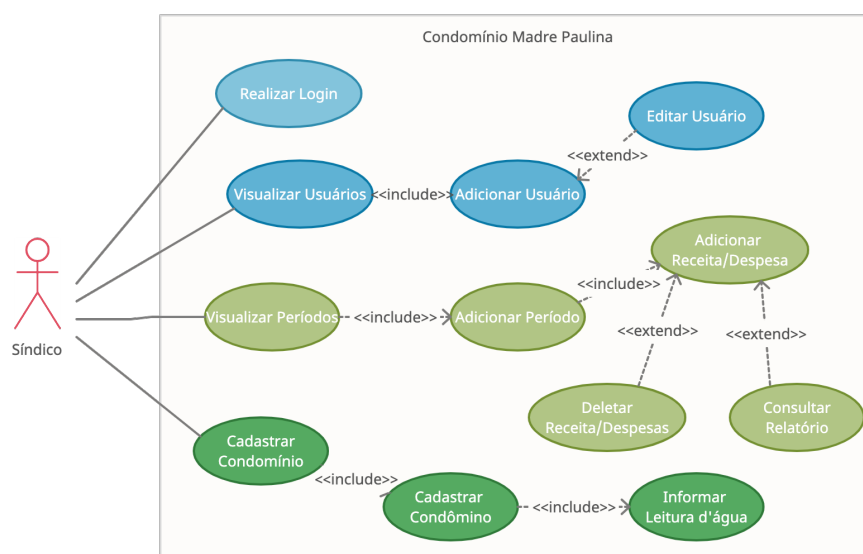


Figura 1. Diagrama Casos de Usos

3. Especificação dos Casos de Usos

A especificação dos casos de usos de uma sistema é um documento funcional, que descreve as funcionalidade que o sistema deve ter, também é apresetando com o sistema deve reagir ao conjunto de dados de entrada que recebe, qual o processamento destes dados e qual a retorno esperado. Dentre os casos de uso do sistema apresentados no diagrama de casos de uso, foram escolhidos alguns para serem detalhados, nas seções abaixo.

3.1. RF001 - Adicionar período de Receitas / Despesas

O usuário na condição de síndico, deve ter a possibilidade de adicionar um novo período. O cadastro deste período deve ser realizado por meio de um formulário, o usuário deve informar um Nome, Data Início, Data fim e Status (Aberto / Fechado). Após a submissão os dados devem ser validados e se tudo estiver correto, então o período deve ser cadastrado e o usuário deve ser redirecionado para a tela de listagem de períodos. Por outro lado, caso os dados informados estejam incorretos, o aplicativo deve exibir uma mensagem de erro e não cadastrar o período.

3.2. RF002 - Adicionar Usuário

O usuário na condição de síndico, deve ter a possibilidade de adicionar um novo usuário. O cadastro desse usuário deve ser por meio de um formulário, o síndico deve informar um Nome, E-mail, Senha, Confirmação de Senha e Perfil. Após a submissão os dados devem ser validados e se tudo estiver correto, então o novo usuário deve ser cadastrado e o usuário deve ser redirecionado para a tela de listagem de usuários. Por outro lado, caso os dados informados estejam incorretos, o aplicativo deve exibir uma mensagem de erro e não cadastrar o novo usuário.

3.3. RF-004 Adicionar receita/despesa

O usuário na condição de síndico, deve ter a possibilidade de adicionar uma nova receita/despesa (lançamentos) em um período. Com um período cadastrado, deve ser possível adicionar uma nova receita/despesa para o respectivo período, esse cadastro deve ser realizado por meio de um formulário, onde devem ser informados uma Descrição, Valor, Data e Tipo (Entrada / Saída). Após a submissão os dados devem ser validados e se tudo estiver correto, então a receita/despesa deve ser adicionada ao período e o síndico deve ser redirecionado para a tela de detalhes do período. Por outro lado, caso os dados informados estejam incorretos, o aplicativo deve exibir uma mensagem de erro e não adicionar a receita/despesa.

O aplicativo também não deve permitir adicionar uma receita/despesa ao período, caso ele esteja com o status *Fechado* e então exibir uma mensagem de alerta e não adicionar a receita/despesa. O mesmo deverá acontecer caso haja uma tentativa de exclusão de uma Receita/Despesa.

3.4. RF-007 Cadastrar Condomínio

O usuário na condição de síndico, deve ter a possibilidade de cadastrar os dados referentes ao seu condomínio. O cadastro deve ser feito realizado por meio de uma formulário, o síndico deve informar, Nome, CNPJ, Valor condomínio 2 quartos, Valor condomínio 3 quartos, Valor Condomínio Sala Comercial, Valor m³ d'água, Taxa de uso sala de festas,

Taxa limpeza salão de festas, Taxa de mudança. Após a submissão os dados devem ser validados e se tudo estiver correto, então o condomínio deve ser cadastrado e o síndico deve ser redirecionado para a tela de listagem de condomínios. Por outro lado, caso os dados informados estejam incorretos, o sistema deve exibir uma mensagem de erro e não cadastrar condomínio. A aplicação também deverá possuir um histórico de valores, pois todo ano há reajustes nos valores cobrados pelo condomínio.

3.5. RF-008 Cadastrar Condômino

O usuário na condição de síndico, deve ter a possibilidade de cadastrar os condôminos do seu condomínio. Com um Condomínio cadastrado, deve ser possível por meio de uma mestre detalhe, adicionar os moradores do condomínio. O síndico deve informar, Número do apartamento, Nome, CPF, Síndico [Sim/Não], Tipo [Apartamento, Sala Comercial], Número de quartos, Ativo [Sim / Não], Ordem. Após a submissão os dados devem ser validados e se tudo estiver correto, então o condômino deve ser cadastrado, caso contrário o sistema deve exibir uma mensagem de erro e não cadastrar o condômino.

4. Desenvolvimento da Aplicação

A partir da Especificação dos Casos de Usos, foi possível dar início ao desenvolvimento da aplicação, o desenvolvimento ficou dividido em duas plataformas, sendo elas um Aplicativo Móvel [4.1] e uma Plataforma WEB [4.2].

4.1. Aplicativo Móvel

Dentre as funcionalidades desenvolvidas para o aplicativo móvel está a interface para a implementação do caso de uso Adicionar Período [3.1]. A partir do cadastro de um novo período, conforme a Figura 2, é possível iniciar o lançamento de Receitas e Despesas conforme o caso de uso Adicionar receita/despesa [3.3], para adicionar um lançamento é necessário informar uma Descrição, Valor, Data e Tipo, conforme Figura 3.

Figura 2. Cadastro de período

Figura 3. Cadastro de receita/despesa

O aplicativo também dispõe de uma tela para que o Síndico possa acompanhar o resumo mensal do condomínio, conforme Figura 4, a tela exibe o montante de entrada e de saída e o respectivo saldo para o respectivo período, esta tela também fornece o histórico de lançamentos para o período.

Outra funcionalidade implementada nesta tela, é a validação para bloquear a adição ou exclusão de um lançamento do período, caso ele esteja com o status *Fechado*, conforme a especificação do caso de uso Adicionar receita/despesa [3.3], a Figura 5 mostra o resultado da implementação deste caso de uso.



Figura 4. Detalhes de um período

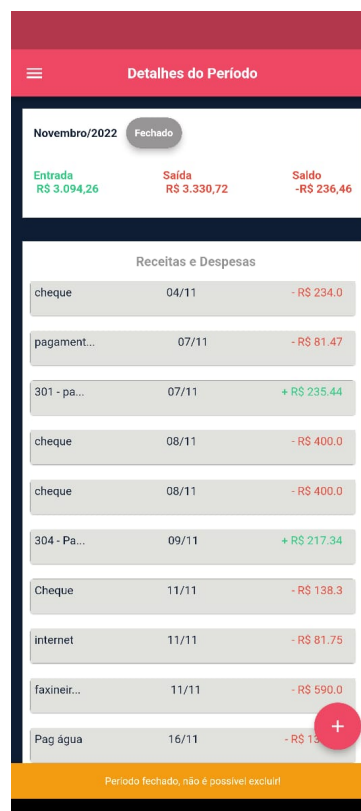


Figura 5. Exclusão de um Detalhe do Período

4.2. Plataforma Web

A plataforma WEB permite ao síndico gerenciar o condomínio e seus respectivos condôminos, nele também é possível gerenciar os valores do condomínio, algumas taxas básicas, como uso do salão de festas, taxa de mudança e outros valores referentes ao m³ da água e valor para a geração do boleto, conforme a Figura 6.

Outra funcionalidade presente na plataforma, é a tela para realizar o gerenciamento da leitura d'água de um período, Figura 7, sendo possível gerenciar o consumo d'água para cada apartamento, por meio de um cálculo automático, que considera o consumo feito na última leitura e acréscimos dos valores referentes às taxas básicas, e se houverem, também podem ser adicionados os valores referentes às taxas de uso de salão de festas e mudanças. Após a geração dos boletos no Internet Banking é possível anexá-los, para que os condôminos possam visualizar os boletos por meio de um relatório.

A plataforma também dispõe um relatório, onde os condôminos podem acessar e acompanhar o seu consumo mensal d'água por meio de um gráfico de barras e também visualizar os boletos do condomínio, respectivamente Figura 8 e Figura 9. Neste relatório caso o usuário possua o perfil *Condômino*, será exibido apenas os dados referentes ao seu

Cód

4

Nome

Condomínio Residencial Madre Paulina

CNPJ

14.829.025/0001-07

Taxa Geração Boleto

2.21

Taxa Básica d'água

35.61

Valor Cond. 2 Quartos

112.02

Valor Cond. 3 Quartos

137.62

Valor Cond. Sala Comercial

83.21

Valor Água

7.5

Taxa Salão de Festas

25

Taxa limpeza Salão de Festas

25

Taxa Mudança

100

Condôminos

Apartamento	Nome	Síndico	Tipo	Número Quartos	Ativo
SALA02	Silvino Bernardo Lamb	Não	Sala Comercial	0	Sim
SALA01	Silvino Bernardo Lamb	Não	Sala Comercial	0	Sim
203	Dian Paulo Silvestre	Não	Apartamento	2	Sim
202	Itacir Nepomuceno	Sim	Apartamento	3	Sim
101	Dariano Fontana	Não	Apartamento	3	Sim
204	Claudete da Silva	Não	Apartamento	2	Sim
201	Rafael Panho	Não	Apartamento	3	Sim

Figura 6. Interface para manutenção do condomínio e condôminos

Cód

54

Condomínio

Condomínio Reside...

Período

Novembro/2022

Data Leitura

06/11/2022

Data Vencimento

22/11/2022

Buscar

► Valor boleto: R\$ 2,21

► Valor Água: R\$ 7,50

► Taxa Básica: R\$ 35,61

► Valor Uso Salão de Festas: R\$ 25,00

► Valor Limpeza Salão de Festas: R\$ 25,00

► Valor Mudança: R\$ 100,00

Condôminos

Apartamento	Consumo		Valor Condomínio	Taxas			Valor Total	Boleto
	Atual	Mês m³		Uso Salão	Limpeza Salão	Mudança		
SALA02 - Silvino Bernardo Lamb	69	0	R\$ 83,21	0	0	0	R\$ 121,03	salas.pdf
SALA01 - Silvino Bernardo Lamb	110	4	R\$ 83,21	0	0	0	R\$ 151,03	
203 - Dian Paulo Silvestre	746	5	R\$ 112,02	0	0	0	R\$ 187,34	

Figura 7. Interface controle da leitura d'água

apartamento, para usuários com o perfil de Síndico e Administrador, existe a possibilidade de aplicar um filtro por condômino.

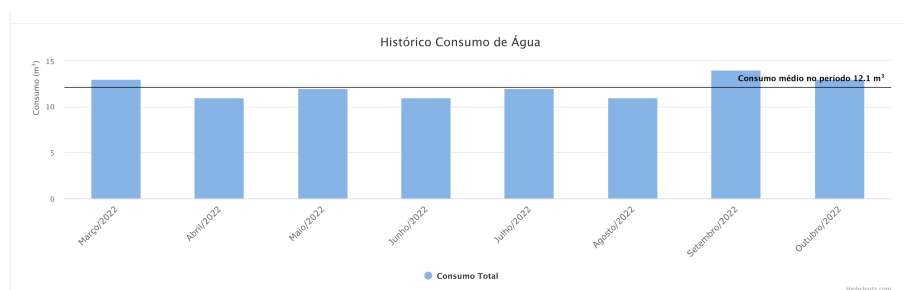


Figura 8. Gráfico consumo d'água

Outro relatório administrativo que está presente na plataforma é o de Relatório de Receitas e Despesas, Figura 10. Neste relatório é possível visualizar as receitas/despesas e o saldo mensal de um ano. Na plataforma também estão disponíveis as funcionalidades referentes a cadastro de usuário e controle de permissões.

Relatório Leitura Água					
Data Inicial	Data Final	Condômino			
01/03/2022	31/12/2022	202 - Itacir Nepomuceno			
✓ Ok	✓ Ok	Filtrar			
Leitura	Condômino	Consumo m³	Dias	Valor	Boleto
Março/2022	202 - Itacir Nepomuceno	13 m³	32	R\$ 106,47	
Abril/2022	202 - Itacir Nepomuceno	11 m³	28	R\$ 120,01	
Maio/2022	202 - Itacir Nepomuceno	12 m³	32	R\$ 100,74	
Junho/2022	202 - Itacir Nepomuceno	11 m³	30	R\$ 95,01	
Julho/2022	202 - Itacir Nepomuceno	12 m³	30	R\$ 127,82	
Agosto/2022	202 - Itacir Nepomuceno	11 m³	29	R\$ 145,32	
Setembro/2022	202 - Itacir Nepomuceno	14 m³	32	R\$ 142,82	
Outubro/2022	202 - Itacir Nepomuceno	13 m³	30	R\$ 135,32	📄

Figura 9. Relatório leitura d'água

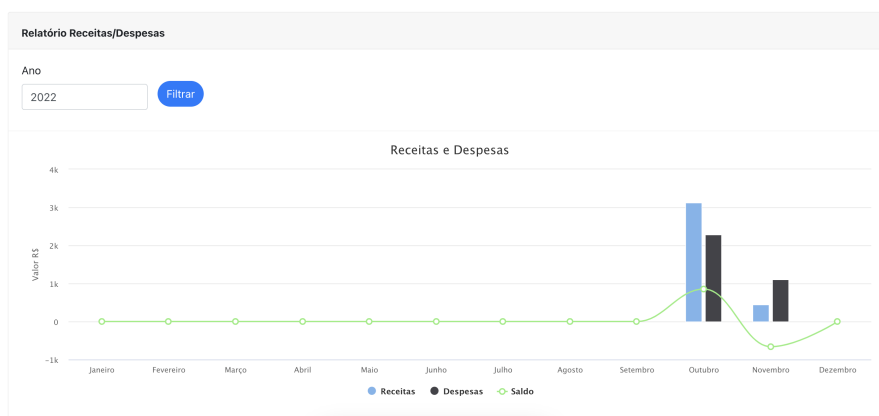


Figura 10. Relatório Receita/Despesas

5. Implementação Técnica da Aplicação

Para garantir uma boa qualidade de código, a aplicação foi desenvolvida seguindo os princípios do Test Driven Development [5.5], que consiste em escrever os testes antes do código. Outro princípio utilizado para desenvolvimento da aplicação foi o SOLID, que consiste em 5 princípios de orientação a objetos, que juntamente com os princípios de Clean Architecture, se seguidos garantem uma boa qualidade de código. As seções abaixo apresentam os detalhes destes princípios aplicados ao projeto.

5.1. Arquitetura

Para o desenvolvimento da aplicação foi adotada arquitetura em camadas, que é um dos padrões arquiteturais mais usados, desde que os primeiros sistemas de software de maior porte foram construídos nas décadas de 60 e 70. Em sistemas que seguem esse padrão, as classes são organizadas em módulos de maior tamanho, chamados de camadas. As camadas são dispostas de forma hierárquica, como em um bolo. Assim, uma camada somente pode usar serviços — isto é, chamar métodos, instanciar objetos, estender classes, declarar parâmetros, lançar exceções, etc. — da camada imediatamente inferior [11, Capítulo 7.2].

Conjuntamente com a arquitetura em camadas, foram aplicados os conceitos de Clean Architecture, conceitos estes definidos por Robert C. Martin em seu livro Clean Architecture: A Craftsman's Guide to Software Structure and Design, publicado em 2017 [6].

5.2. Clean Architecture (Arquitetura Limpa)

Clean Architecture ou também chamada de Arquitetura Limpa é um padrão de projeto de software para arquitetura de software que segue os conceitos de código limpo e implementa princípios SOLID [5.4].

Os padrões de projeto podem ser definidos como boas práticas que ajudam a manter a lógica de negócios unida para minimizar as dependências dentro do sistema. Seguindo a arquitetura limpa permite aos arquitetos de software desacoplar componentes para que fiquem isolados o suficiente para serem duráveis e facilmente alterados sem refazer o sistema.

Dessa forma, seguindo estas boas práticas foi elaborada a proposta de arquitetura para o requisito RF001 - Adicionar um novo usuário [2.1.1], representada através de um diagrama na Figura 11, a arquitetura dos demais casos de uso seguirão a mesma ideia. Os detalhes de cada camada serão explanados na seção [5.3].

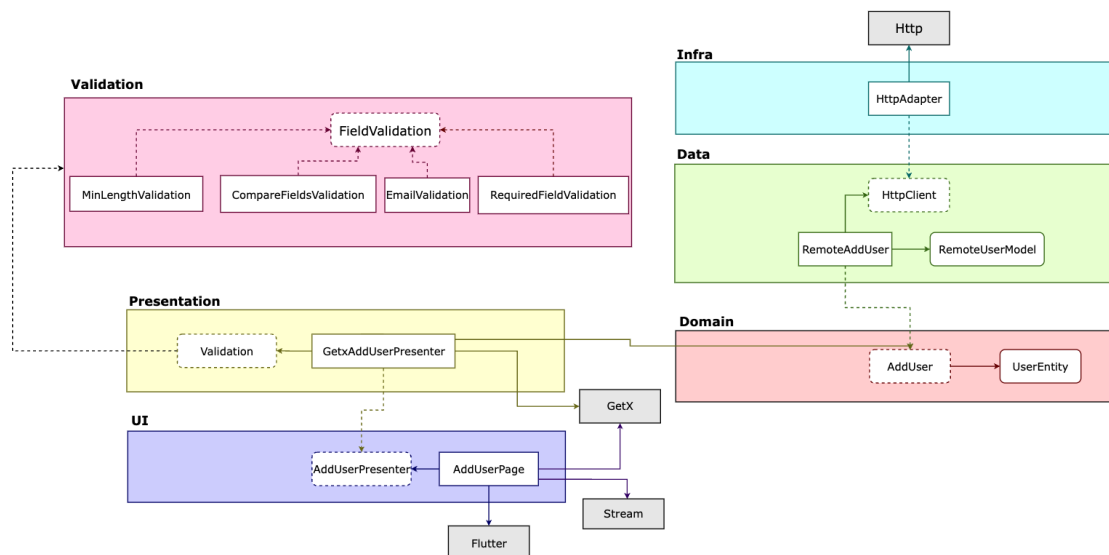


Figura 11. Proposta de Arquitetura para a aplicação.

5.3. Camadas

As camadas (layers) são o núcleo principal para arquitetura limpa. Para o desenvolvimento do aplicativo, foram utilizadas sete camadas: Domínio, Dados, Infraestrutura, Principal, Apresentação, Interface de Usuário, Validação.

5.3.1. Domain Layer (Camada de Domínio)

A camada de domínio é uma abstração responsável por encapsular as regras de negócios do sistema e fornecer os casos de uso. Para manter essas classes simples, cada caso de uso deve ter responsabilidade apenas sobre uma única funcionalidade e não deve conter dados mutáveis. Em vez disso, os dados devem ser manipulados nas camadas de dados [5.3.2]. Utilizando a camada de domínio é possível obter alguns benefícios, entre eles, permitir a divisão de responsabilidade, reduzir a duplicação de código e melhorar a testabilidade do aplicativo.

5.3.2. Data Layer (Camada de Dados)

A camada de dados é responsável por acessar os dados de alguma fonte, seja ela um banco de dados, um arquivo, uma API, etc. Esta camada implementa as regras de negócios definidas pela camada de domínio, ou seja, ela não deve conter nenhuma regra de negócio, apenas implementar as regras definidas pela camada de domínio.

5.3.3. Infra Layer (Camada de Infraestrutura)

Camada de infraestrutura é responsável por se comunicar com serviços externos, como por exemplo, um banco de dados, uma API, ou seja, qualquer serviço externo que a aplicação precise se comunicar, por meio de interfaces. Neste projeto a camada de infraestrutura é responsável por se comunicar com a API e também com a memória cache dos dispositivos.

5.3.4. Main Layer (Camada Principal)

A camada principal é responsável por realizar a composição das demais camadas, por meio de injeção de dependência ¹, sendo também responsável por inicializar a aplicação. Em casos de uma API, esta camada é responsável por inicializar o servidor web e expor as rotas da aplicação.

5.3.5. Presentaion Layer (Camda de Apresentação)

A camada de apresentação é responsável por expor a aplicação para o usuário, seja ela uma API, um site, um aplicativo, etc. Neste projeto, também é a camada responsável por executar as ações do usuário, como por exemplo, receber os dados de um formulário.

Em casos de uma API, esta camada é responsável por receber os dados da requisição, realizar o processamento dos dados e dar um retorno para o usuário após o processamento.

5.3.6. UI Layer (Camada de Interface de Usuário)

A camada de interface de usuário é a camada onde o usuário interage com a aplicação. Neste projeto, a camada de interface é responsável por exibir os dados para o usuário, e também é responsável por receber as ações do usuário, como por exemplo um formulário, um botão, etc.

5.3.7. Validation Layer (Camada de Validação)

A camada de validação é responsável por realizar a validação de dados informados pelo usuário na camada UI. Para o projeto foi implementada a validação de tamanho mínimo

¹ Injeção de dependência é uma técnica para livrar ou remover o acoplamento entre os objetos.

de um campo, e-mail, CPF, senha e confirmação de senha.

5.4. Princípios SOLID

Os princípios SOLID são cinco princípios de programação orientada a objetos e design de código que ajudam a tornar os códigos mais fáceis de entender, manter e estender ². Os princípios SOLID foram definidos por Robert C. Martin em seu livro Clean Architecture [6], os 5 princípios que compõe o SOLID são:

5.4.1. L - Liskov Substitution Principle (Princípio da Substituição de Liskov)

O Princípio de Substituição de Liskov (LSP) diz que objetos podem ser substituídos por seus subtipos sem que isso afete a execução correta do programa. O principal objetivo do LSP é manter o funcionamento do código íntegro no processo de acoplamento de funcionalidades na aplicação. Esse princípio é quebrado em situações nas quais uma sub-classe deixa de herdar um comportamento da classe pai, seja sobrescrevendo um método e lançando uma exceção ou não tirando proveito de todas as funcionalidades dela.

5.4.2. I - Interface Segregation Principle (Princípio da Segregação da Interface)

O Interface Segregation Principle (ISP) nos diz que uma classe não deve ser forçada a implementar uma interface que ela não irá utilizar. O ISP é um princípio que nos ajuda a manter nossas classes mais coesas e focadas em uma única responsabilidade. Um exemplo de ISP é o uso de interfaces ou classes abstratas. Uma interface é um contrato que define um conjunto de métodos que uma classe deve implementar

5.4.3. D - Dependency Inversion Principle (Princípio da Inversão da Dependência)

O Dependency Inversion Principle (DIP) nos diz que os módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações e as abstrações não devem depender de detalhes. Um exemplo de DIP, assim como no ISP é o uso de interfaces, ou seja, uma classe pode depender de uma interface, mas não de uma implementação concreta.

5.5. Test Driven Development

Test Driven Development (TDD), ou Desenvolvimento Orientado a Testes, é uma prática de desenvolvimento de software sugerida por diversas metodologias ágeis, como o Extreme programming. A prática sugere aos desenvolvedores que os testes automatizados sejam escritos de maneira contínua, ao longo do desenvolvimento do software ³. Dessa forma a suíte de testes é maior, abrangendo mais cenários de testes [11, Capítulo 8.1].

Para um melhor aproveitamento do uso do TDD, foram elaborados alguns casos de testes a nível de integração para a camada de dados do requisito funcional RF-003 Adicionar período de Receitas / Despesas [3.3], implementação *RemotAddPeriod*, que

²Exemplos e Aplicação dos conceitos SOLID estão descritos no Apêndice A.

³Mais detalhes sobre os resultados obtidos ao usar o TDD estão no Apêndice B

faz uso da camada de infraestrutura para realizar as requisições para a API, usando a implementação da interface *HttpClient*. Para isso, foram elaborados os seguintes casos de testes:

- *RemotAddPeriod* deve chamar *HttpClient* com os valores corretos.
- *RemotAddPeriod* deve lançar *UnexpectedError* quando *HttpClient* retornar erro 400, 404, 500.
- *RemotAddPeriod* deve retornar um *Period* quando *HttpClient* retornar 200.

Na Figura 12, é possível analisar a implementação de um teste automatizado a nível de integração para a classe *RemotAddPeriod*, nela inicialmente é criado um Spy⁴ para a classe *HttpClient*, este Spy irá armazenar os valores utilizados para realizar a requisição, além de permitir mockar erros, conforme podemos ver na Figura 13. Posteriormente é realizado a chamada do método *add* do SUT (Sistema Sobre Testes)⁵ e por fim é verificado se o método *request* do *HttpClient* foi chamado com os valores corretos.



```
test('Should call HttpClient with correct values', () async {
  httpClient = HttpClientSpy();
  await sut.add(params);

  verify(
    () => httpClient.request(
      url: url,
      method: 'post',
      body: {
        'name': params.name,
        'start_date': params.startDate,
        'end_date': params.endDate,
        'status': params.status,
      },
    ),
  );
});
```

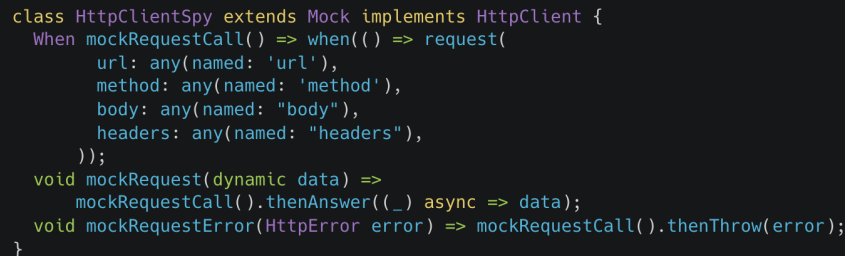
Figura 12. Exemplo de um teste da classe RemoteAddPeriod

Na figura Figura 14 podemos ver um teste automatizado para verificar o retorno esperado para quando a implementação *HttpClient* lançar uma exceção quando ocorrer um erro com o status http 500. Neste teste também é utilizada a classe *HttpClientSpy*, onde é feito um mock de erro do tipo *ServerError* para a request.

Os testes para os demais casos de erros para os status 400 e 404, seguem o mesmo padrão, apenas mudando o tipo do erro utilizando no mock, sendo respectivamente *BadRequest* e *NotFound*.

⁴Spy é uma denominação dada a um objeto que grava suas interações com outros objetos.

⁵Sistema Sobre Testes, refere-se a classe que está sendo testada.

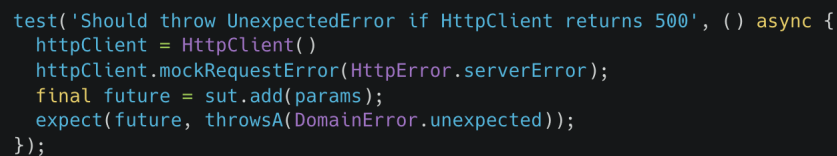


```

class HttpClientSpy extends Mock implements HttpClient {
  when mockRequestCall() => when(() => request(
    url: any(named: 'url'),
    method: any(named: 'method'),
    body: any(named: "body"),
    headers: any(named: "headers"),
  ));
  void mockRequest(dynamic data) =>
    mockRequestCall().thenAnswer((_) async => data);
  void mockRequestError(HttpError error) => mockRequestCall().thenThrow(error);
}

```

Figura 13. Spy para a classe Http Client



```

test('Should throw UnexpectedError if HttpClient returns 500', () async {
  httpClient = HttpClient()
  httpClient.mockRequestError(HttpError.serverError);
  final future = sut.add(params);
  expect(future, throwsA(DomainError.unexpected));
});

```

Figura 14. Exemplo de um teste de erro da classe RemoteAddPeriod

6. Conclusões

O presente projeto teve como objetivo o desenvolvimento de uma plataforma para o condomínio Madre Paulina. Durante o desenvolvimento procurou-se respeitar os requisitos funcionais e não-funcionais levantados com os usuários. Dessa o projeto abre caminho para que o condomínio Madre Paulina e futuramente outros condomínios possam utilizar uma plataforma de análise e planejamento para melhorar tomadas de decisões, reduzindo custos, por exemplo.

Por outro lado, com o desenvolvimento orientado a testes, aliado aos princípios de Clean Architecture e SOLID, acredita-se que será possível adicionar novas funcionalidades para o aplicativo sem muito esforço. A aplicação destes conceitos e princípios também será de grande valia para a carreira profissional do autor, tendo em vista que o mercado da tecnologia da informação está cada vez mais exigente.

7. Trabalhos Futuros

Para o curto prazo, propõe-se realizar a entrega formal do aplicativo e das melhorias realizadas na plataforma ao condomínio Madre Paulina e realizar uma pesquisa com os

usuários, a fim de avaliar a experiência durante o uso da aplicação, bem como elencar possíveis melhorias.

Outro ponto interessante para empreender um esforço, seria o de realizar um estudo sobre plataformas de hospedagem de serviços, considerando que a partir de novembro o Heroku não oferecerá mais serviços gratuitos.

Por fim, sugere-se retomar com os usuários finais as funcionalidades referentes a Reversa do Salão de Festas e a Atas de Assembleia para definir as regras de negócio e posteriormente implementá-las. Também pode-se considerar o desenvolvimento da integração com o Internet Banking para a geração dos boletos, visto que atualmente a geração está sendo feita de forma manual, toranando-se uma atividade morosa.

Referências

- [1] CORPORATION, O. Mysql. <https://www.mysql.com/>. Acesso: 2022-12-08.
- [2] GOOGLE. Angular. <https://angular.io/>. Acesso: 2022-12-08.
- [3] GOOGLE. Code coverage best practices. <https://testing.googleblog.com/2020/08/code-coverage-best-practices.html>. Acesso: 2022-11-22.
- [4] GOOGLE. Dart. <https://dart.dev/>. Acesso: 2022-12-08.
- [5] GOOGLE. Flutter. <https://flutter.dev/>. Acesso: 2022-12-08.
- [6] MARTIN, R. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson, 2017.
- [7] NEPOMUCENO, W. Aplicativo madre paulina. <https://play.google.com/store/apps/details?id=paulina.madre.condominioapp>. Acesso: 2022-12-04.
- [8] NEPOMUCENO, W. Plataforma web madre paulina. <https://madre-paulina.herokuapp.com/>. Acesso: 2022-12-04.
- [9] OTWELL, T. The php framework for web artisans. <https://laravel.com/>. Acesso: 2022-12-08.
- [10] SERVICES, A. W. Amazon relational databases services. <https://aws.amazon.com/pt/rds/>. Acesso: 2022-12-08.
- [11] VALENTE, M. T. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. Independente, 2020.

APÊNDICE

A. Princípios SOLID

Neste apêndice estão alguns exemplos de aplicação dos princípios SOLID, utilizados durante o desenvolvimento da aplicação.

A.1. Princípio da Substituição de Liskov

Neste projeto o LSP foi utilizado para sobrescrever as funcionalidades da classe *HttpAdapter*, para que fosse possível realizar requisições HTTP autenticadas, ambas as classes implementam a interface *HttpClient*. Essa sobrecarga foi feita utilizando o padrão de projeto Decorator, ou seja, é um padrão de projeto que nos permite adicionar novas funcionalidades a um objeto dinamicamente, sem afetar o comportamento de outros objetos do mesmo tipo.

Conforme a Figura 15, a classe *AuthorizeHttpClientDecorator* implementa a interface *HttpClient* e também recebe um objeto do tipo *HttpClient*, chamado *decoratee*, é este objeto que está sendo sobrescrito. No método *request* podemos ver que está sendo adicionado um novo comportamento na classe e posteriormente repassando a ação para a classe pai, neste caso é a classe *HttpAdapter*.



```
class AuthorizeHttpClientDecorator implements HttpClient {
    FetchSecureCacheStorage fetchSecureCacheStorage;
    DeleteSecureCacheStorage deleteSecureCacheStorage;
    HttpClient decoratee;

    AuthorizeHttpClientDecorator({
        required this.fetchSecureCacheStorage,
        required this.deleteSecureCacheStorage,
        required this.decoratee,
    });

    @override
    Future<dynamic> request({ ... }) async {
        try {
            final token = await fetchSecureCacheStorage.fetch('token');
            final authorizedHeaders = headers ?? {}
                ..addAll({'Authorization': 'Bearer $token'});

            return await decoratee.request(
                url: url,
                method: method,
                body: body,
                headers: authorizedHeaders,
            );
        } catch (error) { ... }
    }
}
```

Figura 15. Liskov Substitution Principle

A.2. Princípio da Segregação da Interface

Neste projeto uma aplicação deste princípio é o módulo *SecureStorageAdapter* Figura 17, que é responsável pela comunicação com a memória cache dos dispositivos. Como é possível salvar, ler e deletar os dados da memória cache, foi feita a segregação das ações três

interfaces Figura 16, dessa forma caso os dados precisassem serem obtidos de outro local, seria realizado a modificação apenas dentro do código da implementação da interface.

```
abstract class SaveSecureCacheStorage {
    Future<void> save({required String key, required String value});
}

abstract class FetchSecureCacheStorage {
    Future<String?> fetch(String key);
}

abstract class DeleteSecureCacheStorage {
    Future<void> delete(String key);
}
```

Figura 16. Interfaces SecureStorage

```
class SecureStorageAdapter implements SaveSecureCacheStorage, FetchSecureCacheStorage, DeleteSecureCacheStorage {
    final FlutterSecureStorage secureStorage;

    SecureStorageAdapter({required this.secureStorage});

    @override
    Future<void> save({required String key, required String value}) async {}

    @override
    Future<String?> fetch(String key) async {...}

    @override
    Future<void> delete(String key) async {...}
}
```

Figura 17. Implementação SecureStorageAdapter

A.3. Princípio da Inversão da Dependência

Neste projeto uma aplicação deste princípio é o caso de uso AddAccount, onde é realizada a injeção da classe abstrata HttpClient Figura 18 e não da classe concreta HttpAdapter responsável por realizar as requisições Http Figura 19.

B. Test Driven Development

Aplicando a prática do TDD para desenvolver este projeto foi possível obter uma cobertura de testes de cerca de 85.1%, como mostra a Figura 20. Conforme as melhores práticas de cobertura de código da Google, [3] a cobertura de testes do projeto ficou na faixa "Recomendável", que fica entre 75% a 90%. Por outro lado, as camadas que possuem regras de negócios (data, domain, infra, validation) tiveram uma cobertura de testes de 100%.



Figura 18. Injeção da dependência HttpClient



Figura 19. Interface para o módulo de requisições Http

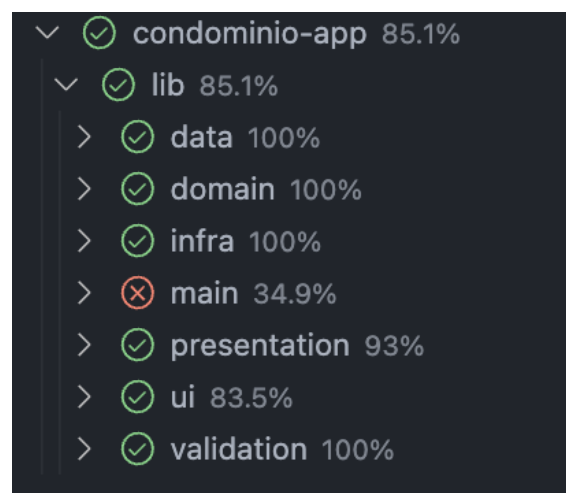


Figura 20. Cobertura de Testes