ISABELLE RODRIGUES COELHO

DESENVOLVIMENTO FULL-STACK NA DTI

LAVRAS – MG

2022

ISABELLE RODRIGUES COELHO

DESENVOLVIMENTO FULL-STACK NA DTI

Relatório de Estágio Supervisionado apresentado à Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel.

Prof. DSc. Ricardo Terra Nunes Bueno Villela

Orientador

LAVRAS – MG

2022

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos

da Biblioteca Universitária da UFLA

Coelho, Isabelle Rodrigues

Desenvolvimento Full-Stack na DTI / Isabelle Rodrigues Coelho. 2^a ed. rev., atual. e ampl. – Lavras : UFLA, 2022.

50 p. : il.

Relatório de estágio (graduação)–Universidade Federal de Lavras, 2022.

Orientador: Prof. DSc. Ricardo Terra Nunes Bueno Villela.

Bibliografia.

1. Desenvolvimento ágil. 2. Sistema Computacional. 3. Áreas de conhecimento.
1. TCC. 2. Monografia. 3. Dissertação. 4. Tese. 5. Trabalho Científico – Normas. I. Universidade Federal de Lavras. II. Desenvolvimento full-stack na DTI.

ISABELLE RODRIGUES COELHO

DESENVOLVIMENTO FULL-STACK NA DTI

Relatório de Estágio Supervisionado apresentado à Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel.

APROVADA em 29 de Abril de 2022.

Profa. DSc. Renata Tales Moreira UFLA
BSc. Douglas Mendes Carmona PUC

Prof. DSc. Ricardo Terra Nunes Bueno Villela
Orientador

LAVRAS – MG
2022

AGRADECIMENTOS

Agradeço primeiramente à minha família, em especial meus pais, Pedro e Cristina, todo apoio e amor durante todas as etapas da minha vida. Aos meus amigos e, em especial, ao meu namorado, Matheus, o companheirismo, o carinho e a torcida.
Ao professor Ricardo Terra, a orientação e disposição. À Universidade Federal de Lavras, em especial ao Departamento de Ciência da Computação e seus profissionais.
À DTI Digital, especialmente aos profissionais da equipe de desenvolvimento na qual atuo.

RESUMO

O documento em questão trata-se de um relatório de um estágio realizado na empresa DTI Digital. Durante o ano de estágio, foi possível aprender e aplicar técnicas de desenvolvimento ágil, bem como imergir na cultura da empresa. Além disso, durante o desenvolvimento e aperfeiçoamento de um sistema computacional, houve o contato com diversas áreas de conhecimento, o que levou à aquisição de conhecimentos de tais âmbitos. Um exemplo se trata da atuação no *back-end* utilizando a linguagem de programação C# e o *framework* .NET. Vale acrescentar que o estágio proporcionou não apenas conhecimento técnico, mas também um crescimento pessoal.

Palavras-chave: Desenvolvimento ágil. Sistema Computacional. Áreas de conhecimento.

LISTA DE FIGURAS

Figura 2.1 – Diagrama *DevOps* 18
Figura 2.2 – Etapas do funcionamento do *front-end* de um sistema computacional 19
Figura 2.3 – Etapas do funcionamento do *back-end* de um sistema computacional 20
Figura 2.4 – Exemplo remodularização *many-to-one* 29
Figura 2.5 – Representação da Arquitetura Hexagonal 31
Figura 2.6 – Arquitetura do Sistema de cinemas 32
Figura 2.7 – Representação da Arquitetura Limpa 33
Figura 3.1 – Tela inicial do sistema 36
Figura 3.2 – Tela de seleção de tipo de documento - Aluguel 37
Figura 3.3 – Tela de seleção de tipo de documento - Compra 38
Figura 3.4 – Abas “Dados do contratante” e “Dados do Fiador” do PDF 40
Figura 3.5 – Rodapé do PDF após a alteração feita 41
Figura 3.6 – Organização do sistema 45

LISTA DE CÓDIGOS

Código 2.1 – ICaixa.cs 24
Código 2.2 – Cedula.cs 24
Código 2.3 – Caixa.cs 26
Código 2.4 – Testes de unidade 27
Código 3.1 – Exemplo de ALTER TABLE em uma tabela utilizando SQL 39
Código 3.2 – Exemplo de SELECT em uma tabela utilizando SQL 44
Código 3.3 – Exemplo de INSERT em uma tabela utilizando SQL 44

SUMÁRIO

1 INTRODUÇÃO 15
2 REVISÃO DA LITERATURA 17
2.1 *DevOps* 17
2.2 Desenvolvimento de Software 19
2.2.1 *Front-End* 19
2.2.2 *Back-End* 19
2.2.3 *Full-Stack* 20
2.3 Scrum 21
2.4 Teste de unidade 23
2.5 Remodularização 28
2.6 SOLID 29
2.7 Arquitetura Hexagonal 30
2.8 *Clean Architecture* 32
3 ATIVIDADES REALIZADAS 35
3.1 Criação de *feature* 35
3.1.1 Discussão 39
3.2 Alterações em *features* 39
3.2.1 Ajustes em PDFs 40
3.2.2 Bloqueio de campos para edição 41
3.2.3 Remodularização de projeto 42
3.2.4 Discussão 42
3.3 Correção de *bugs* 43
3.3.1 Erro ao tentar gerar um documento 43
3.3.2 Listagem de documentos atrasados 44
3.3.3 Inconsistência de dados 45
3.3.4 Discussão 46

4 CONCLUSÃO 47
REFERÊNCIAS 49

1 INTRODUÇÃO

O objetivo deste documento é descrever algumas das atividades realizadas no decorrer do estágio de desenvolvimento *full-stack* na empresa DTI Digital de 01 de abril de 2021 a 31 de março de 2022. A empresa em questão trabalha com soluções computacionais voltadas para empresas parceiras e tem sua sede localizada em Belo Horizonte (MG). Vale acrescentar que também são feitos desenvolvimentos de software voltados para a empresa, como a Plataforma *Round*[1] que, resumidamente, é voltada para a gestão de pessoas e processos dentro da DTI.

Dentre as ferramentas utilizadas pela equipe de desenvolvimento na qual a estagiária participava, podem ser citadas o *framework Scrum* e o *DevOps*, as quais eram gerenciadas através da ferramenta Azure *DevOps*[2], que corroboram para a cultura ágil que a empresa DTI busca seguir. As linguagens de programação utilizadas nos projetos que ficaram sob responsabilidade da equipe de desenvolvimento foram C#, além do *framework* .NET para o *back-end*; e *TypeScript*, juntamente com o *framework* Angular, a linguagem de marcação HTML *(HyperText Markup Language)* e a linguagem de estilos CSS *(Cascading Style Sheets)*, para o desenvolvimento *front-end*. Para as manipulações em banco de dados foi aplicada a linguagem SQL *(Standard Query Language)*.

A equipe em que a estagiária estava alocada era composta por cinco desenvolvedores da DTI, sendo um deles a própria estagiária e o "desenvolvedor líder", que era a referência técnica dentro da equipe. Além disso, havia um *Product Owner* e um *Scrum Master*, ambos funcionários da empresa parceira. De forma sucinta, no dia-a-dia, a estagiária participava de todas as reuniões cabíveis à equipe de desenvolvimento, como o *Planning Poker*, que se trata de avaliar as tarefas a serem realizadas durante a *sprint* e dar uma nota com base na sequência de Fibonacci (LUBIENIECKI, 2021). Vale acrescentar que a estagiária também realizava contribuições para as entregas ao cliente, desenvolvendo tarefas designadas pelo "desenvolvedor líder". Essas e outras atividades compõem os ritos do Scrum, os quais eram todos realizados com pequenas adaptações conforme abordado na Seção 2.3.

Durante o estágio em questão, foram realizadas diversas tarefas, como a criação de *features*, que demandou o desenvolvimento *full-stack*, e a edição de PDFs, que requeriu uma atuação voltada para o *back-end*

[1] https://www.dtidigital.com.br/pocket-case/plataforma-round-marketing-como-propulsor-do-produto/
[2] https://azure.microsoft.com/pt-br/services/devops/

de um projeto. Outra atividade tratada nesse documento é a correção de *bugs*, que levou a uma abordagem mais focada no banco de dados e no atendimento direto ao cliente.

O restante do documento é organizado como a seguir. Na Seção 2, é feito um recorte dos temas abordados no documento como um todo, elaborando uma breve definição para cada elemento com base em referências atuais. Na Seção 3, são abordadas as atividades realizadas durante o estágio em questão. Por fim, na Seção 4, são apresentadas as considerações finais acerca das atividades desempenhadas durante o estágio.

2 REVISÃO DA LITERATURA

Este capítulo trata da fundamentação teórica dos aspectos abordados durante as atividades do estágio. Desse modo, são abordados os conceitos de *DevOps*, desenvolvimento *full-stack*, *Scrum* e teste de software, no contexto da Ciência da Computação.

2.1 *DevOps*

Como o ambiente no qual a estagiária atuava trabalha com práticas *DevOps*, é importante contextualizar tal assunto. Um ponto que deve ser abordado é o fato de que, como afirma Irigoyen (2019), a palavra *DevOps* trata-se da contração dos termos, em inglês, que identificam as equipes envolvidas nas atividades de construção e implantação de software, sendo eles:

- *Development* (Desenvolvimento): equipe responsável pela identificação dos requisitos com o cliente, a análise, o projeto, a codificação e os testes.
- *Operations* (Operações): equipe responsável pela implantação em produção, pelo monitoramento e pela solução de incidentes e problemas.

Como mostra Palma (2021), na atualidade, o mercado brasileiro se mostra cada vez mais competitivo, o que acarreta a necessidade de aumentar a velocidade de produção e a qualidade do que é vendido, seja um produto ou um serviço. Dentro do setor de tecnologia, tal realidade também pode se mostrar presente. Com o objetivo de contornar tal cenário, foi pensada a criação do *DevOps*. Além disso, outro "grande motivador para o movimento de integração *DevOps* é que os departamentos tradicionais de TI são divididos, normalmente, entre duas grandes áreas que costumam ficar isoladas em silos e são orientadas por objetivos conflitantes" (IRIGOYEN, 2019). Tais áreas se tratam do desenvolvimento, responsável pela criação e manutenção do sistema e da operação, cuja tarefa é, resumidamente, garantir a qualidade do sistema e de sua regra de negócios. Assim, é possível ocorrer o cenário descrito em Irigoyen (2019) diante de um problema no sistema:

> "A equipe de Desenvolvimento afirma que o código da aplicação está funcionando perfeitamente e diz que o problema está relacionado com algum componente de infraestrutura. [...]. A equipe de Operação garante que o problema está relacionado com a falta de

> qualidade do desenvolvimento. [...]. As acusações podem durar horas ou dias e geram o sentimento de injustiça e desconfiaça entre as pessoas, o que prejudicará futuras atividades com as mesmas equipes."

Buscando integrar as áreas de atuação supracitadas, de modo a melhorar não apenas o fluxo de atuação de todas as equipes envolvidas em um projeto, mas também o ambiente de trabalho, a cultura *DevOps* foi criada.

Desse modo,

> "para implantar *DevOps* é preciso adotar um conjunto de práticas que vão desde a especificação dos requisitos, passando pelo planejamento e controle do projeto, chegando a adoção de processos automatizados para que a demanda seja entregue da forma rápida segura e com garantia de qualidade." (LUCANIA, 2019).

Tais práticas são, geralmente, representadas no formato de um ciclo como mostrado na Figura 2.1, onde cada parte representa um processo, que se repete ao final da execução completa. A ideia da repetição "infinita" tem como objetivo demonstrar que trata-se de um processo contínuo.

Figura 2.1 – Diagrama *DevOps*

Fonte: https://www.infoq.com/br/articles/papel-qa-devops-breve-historico/

Vale acrescentar que, ainda de acordo com Lucania (2019), o diagrama representado pela Figura 2.1 tem as seguintes etapas: planejamento do que será feito no sistema, desenvolvimento, compilação, testes, criação de uma versão estável do sistema, liberação de tal versão para produção[1], operação e monitoramento do sistema.

[1] Versão do sistema que está em uso pelo usuário final (ROVEDA, 2021).

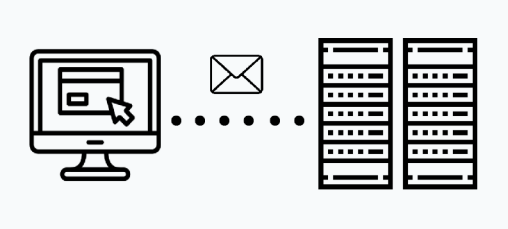
2.2 Desenvolvimento de Software

2.2.1 *Front-End*

O desenvolvimento *front-end*, de acordo com Souto (2019), pode ser definido como a criação, utilizando linguagens de programação adequadas, da interface gráfica que o usuário do sistema interagirá. Além disso, cabe ao desenvolvedor *front-end* realizar a comunicação de tal interface com o *back-end* do sistema, se houver.

A Figura 2.2 exemplifica as etapas do funcionamento de um sistema computacional genérico, no que tange o *front-end*. Como pode ser observado, há uma interface na qual o usuário interage e por onde é realizada a comunicação do *front-end* com o restante do sistema através da troca de mensagens (denominadas "requisições") com o *back-end*, o qual é representado pelas imagens de servidores (elementos mais à direita da figura).

Figura 2.2 – Etapas do funcionamento do *front-end* de um sistema computacional



Fonte: https://www.flaticon.com/br/

2.2.2 *Back-End*

O desenvolvimento *back-end*

> "trabalha em boa partes dos casos fazendo a ponte entre os dados que vem do navegador rumo ao banco de dados e vice-versa, sempre aplicando as devidas regras de negócio, validações e garantias em um ambiente onde o usuário final não tenha acesso e possa manipular algo." (SOUTO, 2019).

A "ponte" supracitada refere-se às APIs (ou *Application Programming Interface*). Uma API desempenha, principalmente, o papel de receber os dados fornecidos pelo navegador, tratando-os conforme as

necessidades da aplicação em questão e enviando-os para o banco de dados da aplicação, como explicado por Shiotsu (2017).

Tal faceta do desenvolvimento de software pode ser exemplificada através da Figura 2.3, que utiliza um sistema genérico como ponto de partida para exemplificar. Conforme ilustrado, a API se comunica com o banco de dados (imagem mais à direita) através do servidor (imagem no centro).

Figura 2.3 – Etapas do funcionamento do *back-end* de um sistema computacional

Fonte: https://www.flaticon.com/br/

2.2.3 *Full-Stack*

Para conceituar o desenvolvimento *full-stack*, é necessário, previamente, revisar os conceitos de desenvolvimento *front-end* e *back-end*. Tal ponto mostra-se importante uma vez que o desenvolvimento *full-stack*, de acordo com Souto (2019), trata-se do desenvolvimento *front-end* atrelado ao desenvolvimento *back-end*.

Tendo em mente as definições supracitadas, é possível compreender, genericamente, o escopo de atuação de um desenvolvedor *full-stack*. Além disso, como afirmado em Northwood (2018), infere-se que o desenvolvedor *full-stack* seja um "generalista especializado", uma vez que podem receber tarefas que englobem diferentes disciplinas ou áreas de conhecimento. Desse modo, é esperado que tal profissional seja capacitado para trabalhar com sistemas completos, não só componentes ou áreas específicas.

A abordagem de tais tipos de desenvolvimento mostra-se relevante, uma vez que a estagiária atuou como desenvolvedora *full-stack*, realizando atividades voltadas para a API e para o banco de dados, além da interface com o usuário.

2.3 Scrum

Tomando como base o contexto atual das empresas de desenvolvimento de sistemas computacionais, tornou-se necessário repensar o processo de desenvolvimento de software. Com isso, propõe-se a utilização do *framework* Scrum como um método para alcançar uma melhor qualidade do que é oferecido pela empresa ou organização que o aplica. Devido a esses fatos, a equipe de desenvolvimento na qual a estagiária atuou, adotou as práticas do Scrum. Assim, mostra-se importante compreender os conceitos que englobam o *framework* em questão.

O Scrum, de acordo com Sutherland (2020), pode ser definido como uma estrutura construída para colocar em prática os seguintes valores levantados no "Manifesto Ágil" Beck (2001):

> "pessoas em vez de processos; produtos que realmente funcionem em vez de documentação dizendo como o produto deveria funcionar; trabalhar com os clientes em vez de negociar com eles; e responder às mudanças em vez de seguir um plano." (SUTHERLAND, 2020).

Vale acrescentar que "o *framework* insere comunicação entre os membros da equipe, verificação contínua do andamento do projeto, e ainda flexibilidade acerca das mudanças" (LIMA, 2016). Trata-se de um processo que, de acordo com Lima (2016), não é padronizado mas fornece a base para que cada organização adicione suas práticas particulares de engenharia e gestão, conforme o cenário no qual estão inseridas.

Entretanto, é necessário destacar as bases nas quais a metodologia do Scrum é consolidada. De acordo com Sutherland (2020), o Scrum possui três papéis principais:

- Scrum *Master*: é o profissional responsável por conduzir a equipe pelas cerimônias e pelos artefatos do Scrum, ajustando-os conforme o cenário da equipe. Além disso, é papel do Scrum *Master* remover obstáculos que possam prejudicar o prosseguimento do trabalho.

- *Product Owner*: tal profissional é responsável por definir as prioridades de trabalho da equipe, além de elaborar o *Product Backlog*, que se trata das atividades que devem ser feitas no projeto que ficou sob a responsabilidade da equipe.

- Equipe de Desenvolvimento: trata-se de um grupo de profissionais que realizarão o trabalho definido. Esse grupo deve contar com as competências necessárias para o desenvolvimento das atividades, caso contrário devem ser capacitadas para tanto.

Um ponto que merece destaque trata-se das cerimônias do Scrum. Tais eventos são de crucial importância para a aplicação do *framework* em uma organização, uma vez que impactam diretamente na maneira como é feita a abordagem do produto e o atuação no mesmo.

> "No Scrum, os produtos são desenvolvidos em iterações chamadas de *Sprints*, que usualmente compreendem o período de uma semana a um mês. Toda *Sprint* é iniciada por uma reunião formal chamada *Sprint Planning* e encerrada com duas outras reuniões: a *Sprint Review* e a *Sprint Retrospective*." (LEUCOTRON, 2021).

Segundo Sutherland (2020), a *Sprint Planning* trata-se de uma reunião para planejar a *Sprint*. A equipe decide a quantidade de trabalho que acredita ser capaz de realizar no tempo da *Sprint*. Eles escolhem as tarefas no *Product Backlog* e organizam-as em cartões ou *post-its* (criando o chamado *Sprint Backlog*), por exemplo, para poderem discutir e definir o que será feito. Ainda segundo o mesmo autor, durante a *Sprint Review*, a equipe reúne e mostra o que conseguiu fazer naquele tempo, o que promove uma reflexão a respeito da quantidade de tarefas escolhidas e a qualidade do que foi entregue. Já a *Sprint Retrospective*

> "é uma reunião ao final da *Sprint* em que a equipe e o Scrum *Master* se reúnem para autoavaliação, inspeção do próprio Scrum e realização de ajustes. É um encontro importante para a motivação da equipe e a redução do atrito nas relações de trabalho." (LEUCOTRON, 2021)

Vale acrescentar que, durante a *Sprint*, a equipe tem um momento diário pré-definido, com duração média de quinze minutos, para que cada membro de tal equipe possa realizar um pequeno *feedback* de suas atividades. Tal momento é chamado de *Daily Scrum* e, como afirma Leucotron (2021), visa que cada participante responda às seguintes perguntas:

- O que foi feito até o momento da tarefa que está atuando?
- O que objetiva realizar no dia de hoje?
- Há algum obstáculo que impeça o progresso?

Com tal reunião, pode-se ter uma noção do andamento das tarefas, bem como identificar impedimentos e realizar uma preparação para atuar nos mesmos.

Tendo os pontos supracitados em mente, é possível afirmar que a equipe de desenvolvimento na qual a estagiária atuou realizava a maioria dos ritos citados, adaptando apenas a *Sprint Review* para que ocorresse durante o próprio refinamento técnico das tarefas que seriam incluídas na *Sprint*.

2.4 Teste de unidade

Uma etapa essencial no desenvolvimento de sistemas computacionais trata-se da realização dos testes de software. Esse tipo de teste é

> "uma atividade dinâmica. Seu intuito é executar o programa ou modelo utilizando algumas entradas em particular e verificar se seu comportamento está de acordo com o esperado. Caso a execução apresente resultados não especificados, dizemos que um erro ou defeito foi identificado." (DELAMARO, 2007).

Dentre os diversos tipos de testes de software, há o teste de unidade. Tal teste é caracterizado por

> "analisar funcionalidades simples e de resultado constante. Como exemplo, tem-se: uma função de soma entre dois números, o resultado da soma de 1 e 2 deve sempre ser 3. Esse teste é muito importante para assegurar que a "base" do sistema, que será utilizado nas requisições, está em perfeito funcionamento." (FERNANDES, 2018).

Para fins de estudo, pode ser plausível a realização de tais testes manualmente. Entretanto, organizações, geralmente, possuem projetos robustos, logo mostra-se mais interessante automatizar tais testes. De acordo com Sommerville (2011), testes automatizados possuem as seguintes três etapas básicas em sua construção e execução:

- Uma parte de configuração, em que você inicia o sistema com o caso de teste, ou seja, as entradas e saídas esperadas.
- Uma parte de chamada, quando você chama o objeto ou método a ser testado.
- Uma parte de afirmação, em que você compara o resultado da chamada com o resultado esperado. Se a afirmação for verdadeira, o teste foi bem-sucedido; se ela for falsa, ele falhou.

Dentro da organização na qual a estagiária trabalhou foram implementados testes de unidade automatizados voltados para o *back-end* das aplicações, sendo que a estagiária desenvolveu alguns desses testes.

Com o objetivo de demonstrar a implementação de testes de unidade para um sistema fictício simples, tem-se a implementação de "uma aplicação computacional que simula um caixa eletrônico e a disponibilização de saque de acordo com o valor solicitado e as cédulas disponíveis para saque" (BACK, 2020). A linguagem de programação escolhida foi C# e, para realizar os testes de unidade, foi utilizado o *framework* xUnit. Vale acrescentar que o projeto não é de autoria própria, sendo obtido da plataforma GitHub de terceiros, cuja referência se encontra abaixo de cada código nesta subseção.

Código 2.1 – ICaixa.cs

```
using System.Collections.Generic;

namespace CaixaEletronico.Domain
{
    public interface ICaixa
    {
        ICollection<int> Saque(int valor);
        bool ValidaCedulasDisponiveis(int valor);
    }
}
```

Fonte: https://gist.github.com/jozimarback/c95225dbf82d93f0dd626af77aca2dac#file-caixa-cs

Como mostra o Código 2.1, o arquivo "ICaixa.cs" conta com a implementação de uma interface "ICaixa", que possui os métodos *Saque* e *ValidadeCedulasDisponiveis*, sendo que ambas recebem um valor inteiro que corresponde ao valor que deseja-se sacar de uma determinada conta.

Código 2.2 – Cedula.cs

```
namespace CaixaEletronico.Domain
{
    public static class Cedula
    {
        public static int Cem => 100;
        public static int Cinquenta => 50;
        public static int Vinte => 20;
        public static int Dez => 10;
    }
}
```

Fonte: https://gist.github.com/jozimarback/c95225dbf82d93f0dd626af77aca2dac#file-caixa-cs

No Código 2.2, o arquivo “Cedula.cs” conta com a implementação da classe *Cedula*. Na classe em questão, são definidos valores inteiros para cada constante, as quais recebem, respectivamente, os seguintes nomes e valores:

- o valor inteiro cem (100) é atribuído à constante *Cem*.
- o valor inteiro cinquenta (50) é atribuído à constante *Cinquenta*.
- o valor inteiro vinte (20) é atribuído à constante *Vinte*.
- o valor inteiro dez (10) é atribuído à constante *Dez*.

Já no Código 2.3, o arquivo “Caixa.cs” armazena a implementação da classe *Caixa*. Tal classe herda da interface “ICaixa.cs”. Desse modo, a classe em questão implementa os métodos definidos na interface citada anteriormente. Para cada método, há a seguinte lógica:

- Método *Saque*: é criada uma lista (inicialmente vazia), denominada *cedulasSacadas*, para armazenar cada cédula sacada e, a partir do valor recebido por parâmetro, uma variável de nome *valorRestanteASerSacado* recebe tal valor. Após a inicialização de tais elementos, é utilizada a repetição *while* para que, enquanto determinada condição for verdadeira, sejam feitos os passos dentro de seu escopo (do *while*). Desse modo, a variável *valorRestanteASerSacado* é analisada com o objetivo de averiguar se a mesma possui valor maior ou igual a determinada constante pertencente à classe “Cedula.cs”, seguindo a ordem *Cem, Cinquenta, Vinte* e *Dez*. Cada vez que a comparação é positiva, isto é, o valor de *valorRestanteASerSacado* é maior ou igual a determinada constante, é adicionada à lista *cedulasSacadas* o valor (por exemplo, 100) e é subtraído o valor da cédula em questão da variável *valorRestanteASerSacado*. Quando a condição não é satisfeita, finaliza-se o escopo do *while* em questão e passa-se para o próximo. Após todos os *loop*s serem “visitados”, é utilizada a condicional *if* para analisar se a lista *cedulasSacadas* está vazia. Se estiver, é retornada uma exceção com a seguinte mensagem: “Não há cedulas disponíveis para o valor solicitado” e o método é finalizado. Se *cedulasSacadas* não estiver vazia, é retornada tal lista e o método se encerra.

Código 2.3 – Caixa.cs

```
using System.Collections.Generic;

namespace CaixaEletronico.Domain
{
    public class Caixa : ICaixa
    {
        public ICollection<int> Saque(int valor)
        {
            var cedulasSacadas = new List<int>();
            int valorRestanteASerSacado = valor;
            while (valorRestanteASerSacado >= Cedula.Cem)
            {
                cedulasSacadas.Add(Cedula.Cem);
                valorRestanteASerSacado = valorRestanteASerSacado - Cedula.Cem;
            }
            while (valorRestanteASerSacado >= Cedula.Cinquenta)
            {
                cedulasSacadas.Add(Cedula.Cinquenta);
                valorRestanteASerSacado = valorRestanteASerSacado - Cedula.Cinquenta;
            }
            while (valorRestanteASerSacado >= Cedula.Vinte)
            {
                cedulasSacadas.Add(Cedula.Vinte);
                valorRestanteASerSacado = valorRestanteASerSacado - Cedula.Vinte;
            }
            while (valorRestanteASerSacado >= Cedula.Dez)
            {
                cedulasSacadas.Add(Cedula.Dez);
                valorRestanteASerSacado = valorRestanteASerSacado - Cedula.Dez;
            }
            if (cedulasSacadas.Count == 0)
                throw new System.Exception("Nao ha cedulas disponiveis para o valor
                    solicitado.");
            return cedulasSacadas;
        }
        public bool ValidaCedulasDisponiveis(int valor)
        {
            return valor % 10 == 0;
        }
    }
}
```

Fonte: https://gist.github.com/jozimarback/c95225dbf82d93f0dd626af77aca2dac#file-caixa-cs

- Método *ValidaCedulasDisponiveis*: a partir do valor recebido por parâmetro, é feita a divisão desse valor recebido por dez (10), sendo retornada a comparação do resto de tal divisão com zero. Isto é, retorna "*true*" se for igual a zero e "*false*" caso contrário.

Com o objetivo de realizar dois testes de unidade do projeto supracitado, foi feita a implementação da classe de teste *NormalAssertsTest* que consta no Código 2.4. Antes, deve-se ter em mente que

> "com xUnit, a criação de um teste simples pode ser feita com um método adicionando sobre ele o atributo *Fact*. Para assegurar que o teste deve obedecer algum resultado esperado utilizamos a classe *Assert*. Na classe *Assert*, temos uma grande lista de possibilidades, como verificar se resultado é falso, verdadeiro, igual, maior, menor, nulo e até se deve esperar alguma exceção." (BACK, 2020).

Código 2.4 – Testes de unidade

```
using CaixaEletronico.Domain;
using System.Collections.Generic;
using System;
using Xunit;

namespace Training_Tests.UnitTests
{
    public class NormalAssertsTest
    {
        private readonly Caixa caixa = new Caixa();
        [Fact]
        public void Saque_Valido()
        {
            int valorSaque = 510;
            bool saqueValido = caixa.ValidaCedulasDisponiveis(valorSaque);
            Assert.True(saqueValido);
        }
        [Fact]
        public void Deve_Gerar_Excecao()
        {
            int valorSaque = 5;
            Assert.Throws<Exception>(() => caixa.Saque(valorSaque));
        }
        [Fact]
        public void Deve_Retornar_Lista_De_Cedulas_Sacadas()
        {
            int valorSaque = 310;
            List<int> listaEsperada = new List<int>() { 100, 100, 100, 10 };
            List<int> resultado = caixa.Saque(valorSaque);
            Assert.Equal(listaEsperada, resultado);
        }
    }
}
```

Fonte: https://gist.github.com/jozimarback/067b1c9768ead3a09a569750e2b59b96#file-xunit-facts-cs (ADAPTADO com alterações em nomes de variáveis e adicionado o método *Deve_Retornar_Lista_De_Cedulas_Sacadas*)

A partir das informações supracitadas, no Código 2.4 é instanciado um objeto do tipo *Caixa* cujo nome é "caixa". Além disso, há os seguintes métodos de teste:

- Método *Saque_Valido*: a variável *valorSaque* recebe o valor 510 (quinhentos e dez) e a variável *saqueValido* recebe o resultado da chamada do objeto "caixa" ao método *ValidaCedulasDisponiveis*,

passando o valor da variável *valorSaque*. Em seguida, é utilizado o método *Assert.True* para analisar se o valor da variável *saqueValido* é verdadeiro. Se for, o teste foi concluído com sucesso. Caso contrário, houve falha.

- Método *Deve_Gerar_Excecao*: é declarada a variável *valorSaque* recebendo o valor 5 (cinco). A seguir, é utilizado o método *Assert.Throws<Exception>* para analisar o resultado obtido da chamada do método *ValidaCedulasDisponiveis* passando o valor de *valorSaque*. Se uma exceção for lançada, o teste terá êxito. Caso outro resultado seja encontrado, o teste falha.

- Método *Deve_Retornar_Lista_De_Cedulas_Sacadas*: é declarada a variável *valorSaque* recebendo o valor 310 (trezentos e dez) e a lista *listaEsperada* com os valores 100, 100, 100 e 10 (as três primeiras posições de lista contendo o valor cem e a quarta posição contendo o valor dez). A seguir, é utilizado o método *Assert.Equal* para comparar o valor da lista *listaEsperada* e o resultado obtido da chamada do método *Saque* passando o valor de *valorSaque*. Se o resultado da comparação for verdadeiro, o teste terá êxito. Caso outro resultado seja encontrado, o teste falha.

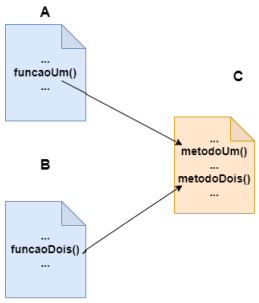
2.5 Remodularização

Uma das atividades realizadas durante o período de estágio tratou-se da remodularização de uma das aplicações que eram responsabilidade da equipe de desenvolvimento. Desse modo, torna-se importante introduzir o conceito de remodularização. Primeiramente, é necessário definir o termo "modularização". De acordo com Junior (2011), modularização "consiste em decompor um programa em uma série de subprogramas individuais". Assim, pode-se afirmar que módulos, no contexto em questão, se trata de um subprograma que pode possuir dados de entrada e dados de saída.

Com o conceito de modularização em mente, tem-se que a remodularização como a "extração de módulos de fragmentos de código relacionados à manipulação de grupos de classes bem definidos" (TERRA et al., 2012) e pode ter três objetivos principais, sendo eles correção, evolução ou adaptação do código. Ainda de acordo com Terra et al. (2012), um exemplo trata-se da estratégia *many-to-one* (ou muitos-para-um), a qual os módulos extraídos são agrupados em uma única classe nova. Esse modelo é mais indicado para

situações nas quais os módulos extraídos raramente utilizam atributos. A Figura 2.4 ilustra um exemplo de remodularização onde dois métodos são adicionados a um novo arquivo de nome A e são referenciadas pelos respectivos métodos que os utilizam.

Figura 2.4 – Exemplo remodularização *many-to-one*



Fonte: Do autor (2022)

Desse modo, a remodularização pode ser aplicada em diferentes cenários, como a refatoração de sistemas ou a mudança de arquitetura de software de um projeto. Vale acrescentar que, para cada tipo de cenário, uma estratégia pode ser melhor indicada que outra, como o caso da metodologia *many-to-one*.

2.6 SOLID

Os princípios SOLID se tratam de cinco "regras" da programação orientada a objetos que, quando aplicadas, "podem facilitar o desenvolvimento de software, tornando-os fáceis de manter e estender" (PAIXÃO, 2019). Ainda de acordo com Paixão (2019), essas regras podem ser explicadas, resumidamente, da seguinte forma:

- *S* ou Princípio da responsabilidade única: uma classe deve possuir apenas um objetivo. Isso evita a criação de *God classes*, as quais não "fazem de tudo", dificultando a compreensão e manutenção do código.

- *O* ou Princípio Aberto-Fechado: quando novos comportamentos e recursos precisam ser adicionados no software, deve-se estender e não alterar o código fonte original. Desse modo, o que já foi implementado não deverá passar por uma refatoração, o que evita erros de lógica.

- *L* ou Princípio da substituição de Liskov: uma classe derivada deve ser substituível por sua classe base. Esse princípio estimula a melhor estruturação das abstrações, além do uso de injeção de dependência, acarretando em um código de melhor compreensão e manutenção.
- *I* ou Princípio da segregação da interface: uma classe não deve ser forçada a implementar interfaces e métodos que não vai utilizar. Assim, as interfaces criadas serão condizentes com o objetivo da(s) classe(s) que a importa(m), podendo ser alteradas com mais facilidade, se necessário.
- *D* ou Princípio da inversão da dependência: um módulo de alto nível não deve depender de módulos de baixo nível e ambos devem depender da abstração. Isto é, as classes devem estar desacopladas e dependendo de uma abstração, de modo a viabilizar alterações no código e na regra de negócios sem impactar o sistema por completo.

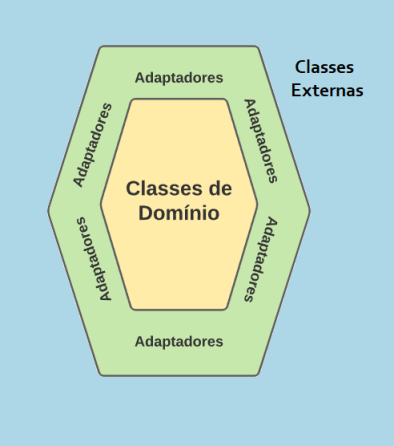
Levando em consideração os elementos supracitados, o SOLID se mostra uma ferramenta importante para o desenvolvimento de sistemas computacionais, uma vez que corrobora para a criação de um padrão de desenvolvimento e para a realização de manutenções nesses sistemas. Tendo isso em mente, a equipe de desenvolvimento na qual a estagiária estava inserida optou por seguir os princípios SOLID durante o processo de desenvolvimento nas aplicações.

2.7 Arquitetura Hexagonal

Dentre os sistemas nos quais a estagiária atuou, a abordagem de uma aplicação *back-end* cuja arquitetura de software se tratava da arquitetura hexagonal proveu bastante aprendizagem acerca das regras de negócio do cliente e da própria arquitetura. Assim, torna-se clara a necessidade de conceituar a arquitetura em questão.

A arquitetura de software hexagonal tem como objetivo "construir sistemas que favorecem reusabilidade de código, alta coesão, baixo acoplamento, independência de tecnologia e que são mais fáceis de serem testados" (VALENTE, 2020). Em uma arquitetura hexagonal, as classes são divididas em dois grupos principais, sendo eles "Classes de domínio" e as classes que envolvem diretamente acesso ao banco de dados, infraestrutura e integração com demais sistemas (se houver). Vale acrescentar que as classes de domínio são aquelas diretamente relacionadas às regras de negócios do projeto.

Figura 2.5 – Representação da Arquitetura Hexagonal



Fonte: https://engsoftmoderna.info/artigos/arquitetura-hexagonal.html

Como mostra a Figura 2.5, para que os dois tipos de classe previamente citados se comuniquem, é necessário utilizar os chamados "adaptadores" que, como afirma Valente (2020),

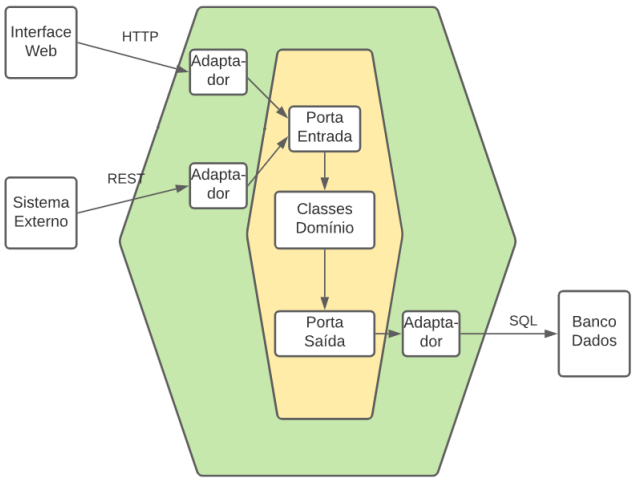
> "atuam recebendo chamadas de métodos vindas de fora do sistema e encaminhando essas chamadas para métodos adequados das portas de entrada, além de também receberem chamadas advindas das classes de domínio e as direcionam para o respectivo sistema externo."

Para ilustrar esse tipo de arquitetura, pode ser utilizado um sistema fictício de cinemas. Vale acrescentar que tal sistema foi pensado com base no exemplo de um sistema de bibliotecas mostrado por Valente (2020).

Como ilustra a Figura 2.6, o sistema de cinemas supracitado pode ser acessado através da interface web e de um sistema externo. Independente da forma de acesso, esse é mediado por adaptadores que se comunicam com uma porta de entrada. As portas de entrada definem métodos de pesquisa no catálogo de filmes, de modo a permitir a compra e a reserva de ingressos, cadastro de usuários, entre outros. Tais métodos são implementados pelas classes de domínio, as quais incluem "Filme", "Ingresso" e "Usuario".

Um ponto importante é o fato de que o sistema deve persistir alguns dados, como o *login* do usuário e a compra de ingressos. Para isso, é utilizada uma porta de saída que implementa métodos para salvar e buscar dados. Um adaptador realiza a tarefa de intermediar a "comunicação" entre tais métodos e um banco de dados relacional.

Figura 2.6 – Arquitetura do Sistema de cinemas



Fonte: https://engsoftmoderna.info/artigos/arquitetura-hexagonal.html

Outra observação a ser feita, trata-se de que

> "um sistema pode possuir várias portas de entrada e de saída (sempre localizadas no hexágono interior, junto às classes de domínio). Em uma determinada porta, seja ela de entrada ou de saída, podemos plugar um ou mais adaptadores, os quais ficam sempre localizados no hexágono mais externo." (VALENTE, 2020).

Assim, pode-se citar que a arquitetura hexagonal possui como uma de suas principais características a organização do sistema computacional em classes internas e classes externas que se comunicam através de adaptadores.

2.8 *Clean Architecture*

Uma das primeiras atividades desenvolvidas durante o estágio se tratou da remodularização de uma aplicação com arquitetura hexagonal para a *Clean architecture* (ou Arquitetura Limpa). Desse modo, é necessário conceituar a arquitetura limpa.

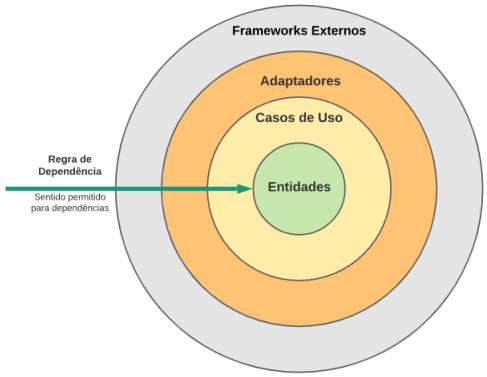
De acordo com Zarelli (2020), a *clean architecture* busca fornecer uma metodologia a ser utilizada na codificação, a fim de facilitar o desenvolvimento de código e promover uma melhor manutenção, além de buscar gerar menos dependências. Outro ponto importante é o fato de que tal arquitetura busca "fornecer

aos desenvolvedores uma maneira de organizar o código de forma que encapsule a lógica de negócios, mas mantendo-o separado do mecanismo de entrega" (ZARELLI, 2020).

Como é possível perceber na Figura 2.7, o sistema é desenvolvido em camadas, sendo elas: entidades, ou *entities*; casos de uso, ou *use cases*; adaptadores, ou *adapters*; e *frameworks* externos. Vale acrescentar que tal tipo de estratégia traz diversas vantagens, uma vez que

> "a separação de camadas poupará o desenvolvedor de muitos problemas futuros com a manutenção do software, a regra de dependência bem aplicada deixará seu sistema completamente testável. Quando um *framework*, um banco de dados ou uma API se tornar obsoleta, a substituição de uma camada não será uma dor de cabeça, além de garantir a integridade do *core* do projeto." (ZARELLI, 2020).

Figura 2.7 – Representação da Arquitetura Limpa



Fonte: https://engsoftmoderna.info/artigos/arquitetura-limpa.html (ADAPTADO)

Explorando mais cada camada da Figura 2.7, tem-se:

- Entidades (ou *entities*): são classes "base" que representam os elementos pertencentes ao sistema, por exemplo, "Usuario", "Carrinho" e "Produto".
- Casos de uso (ou *use case*s): tratam-se de classes que implementam as regras de negócio do sistema utilizando objetos derivados das classes "entities". Tomando um sistema de compras simples para ilustrar, um caso de uso seria que um e-mail não pode ter mais que um usuário relacionado a ele. Assim, dentro da classe que trata tal tipo de caso de uso (por exemplo, "CadastroUsuarioUseCase", a qual seria responsável por atuar nos casos de uso envolvendo o cadastro de um usuário no sistema), seria feita a implementação da análise do atributo e-mail, vinculado a um objeto do tipo "Usuario".

- Adaptadores (ou *adapters*): são classes e/ou interfaces que têm como objetivo intermediar a comunicação entre os sistemas externos e as camadas “Entidades” e “Casos de Uso”. Seguindo exemplo anterior, seria necessário implementar um adaptador para que fosse feita uma consulta no banco de dados da aplicação, de modo a obter os e-mails armazenados e retorná-los para o *use case* que os solicitou.

- *Frameworks* externos: tratam-se das classes de bibliotecas, *frameworks* e outros sistemas utilizados na aplicação. Por exemplo, o sistema que implementa a interface com o usuário.

Um ponto que deve ser ressaltado trata-se do fato de que as classes pertencentes às camadas mais internas da arquitetura não devem “conhecer” as camadas mais externas, como afirmado por Martin (2017). Tal regra de dependência se aplica à funções, classes, variáveis e qualquer outro elemento de código. Assim, “tal regra de dependência garante que entidades e casos de uso sejam classes ‘limpas’ de qualquer tecnologia ou serviço externo ao sistema que foram implementados” (VALENTE, 2020), permitindo que o sistema seja mais adaptável a mudanças.

Tendo em mente os pontos supracitados, percebe-se que a arquitetura limpa engloba diversos conceitos em sua especificação, como propriedades de projeto (por exemplo, coesão, acoplamento e separação de interesses) e princípios de projeto (por exemplo, a responsabilidade única, que é também abordada na Seção 2.6). Além disso, a arquitetura em questão traz diversos desafios como a capacidade de abstrair os elementos do “mundo real” em *entities* que farão parte da base do sistema.

3 ATIVIDADES REALIZADAS

Este capítulo descreve as principais atividades realizadas durante o estágio. Vale acrescentar que tais atividades foram realizadas nos sistemas sob responsabilidade da equipe de desenvolvimento.

Na Seção 3.1, são tratadas as *features* criadas nos sistemas *back-end* e *front-end*. Já na Seção 3.2, são abordados os aspectos referentes às correções de *features* já pertencentes aos projetos tratados na Seção 3.1, além de outro projeto legado. Por fim, a Seção 3.3 apresenta a atuação em *bugs* e incidentes envolvendo essas aplicações computacionais.

Um ponto que vale ser citado trata-se das tecnologias empregadas nos projetos abordados na Seção 3. Para o desenvolvimento *front-end*, a linguagem de programação usada foi TypeScript e o *framework* Angular, juntamente com a linguagem de marcação HTML (*HyperText Markup Language*) e a linguagem de estilos CSS (*Cascading Style Sheets*). A atuação no *back-end* foi predominantemente realizada utilizando a linguagem de programação C# e o *framework* .NET. O banco de dados de tal sistema segue o modelo relacional[1] e é manipulado através da linguagem SQL (*Standard Query Language*).

É necessário elucidar que os detalhes de implementação não foram citados devido ao contrato de confidencialidade assinado com a empresa na qual se decorreu o estágio. No entanto, para passar uma ideia do que de fato foi desenvolvido, criou-se um sistema simples de aluguel de imóveis utilizando as ferramentas de desenvolvimento supracitadas.

3.1 Criação de *feature*

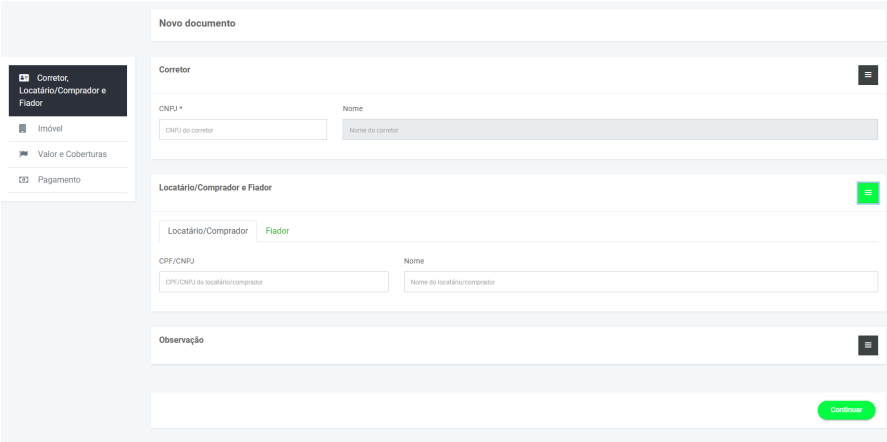
Dentre as *features* criadas durante o período do estágio, tem-se a adição de um novo fluxo de criação e geração de documentos referentes às regras de negócio do cliente. Tal fluxo deveria ser implementado no sistema em que os usuários realizam suas atividades, bem como no *back-end* da aplicação em questão. Tomando o sistema-exemplo criado, havia apenas a possibilidade de criar documentos de aluguel de imóvel. Com a nova *feature* implementada, se tornou possível também gerar documentos de compra de imóvel.

De forma sucinta, era necessário alterar as telas do sistema de modo a comportar novos campos, além de atualizar campos já existentes para que contivessem elementos do novo fluxo. O fluxo anterior possuía

[1] Tipo de banco de dados que armazena e fornece acesso a pontos de dados relacionados entre si. (NETO, 2011)

apenas uma opção de tipo de documento, a qual afetava os outros campos que deveriam ser preenchidos para que o documento fosse gerado com sucesso. Alguns dos dados a serem informados eram referentes às datas de início e término e o valor do contrato, além do endereço do imóvel. Como ilustra a Figura 3.1, o campo do "Corretor" permaneceu inalterado. Entretanto, as abas abaixo foram alteradas de modo a comportar clientes do tipo "Locatário" e do tipo "Comprador".

Figura 3.1 – Tela inicial do sistema



Fonte: Do autor (2022)

Com as mudanças implementadas, foi possível que o usuário selecionasse dentre as duas opções de criação de documento e, a partir da opção escolhida, os campos com mais especificações eram carregados de modo a se adequarem às regras de cada opção. Vale acrescentar que haviam campos "em comum" entre as opções, de modo que esses campos não eram recarregados ao alterar a opção marcada. Para realizar o controle de qual opção era selecionada e, por consequência, quais campos deveriam ser mostrados, foram utilizados os recursos de *Observables*, *Inputs* e *Outputs* do Angular[2] de modo que todos os componentes envolvidos no componente "pai" conseguiam "monitorar" o estado das variáveis que necessitavam para aplicar as devidas validações e mostrar os dados na tela corretamente.

De maneira resumida, um *Input* permite que um componente "pai" atualize dados em um ou mais componentes "filhos". Já um *Output* possibilita que um componente "filho" envie dados para um componente "pai". No contexto da aplicação em questão, quando o usuário selecionava o tipo de documento, o método

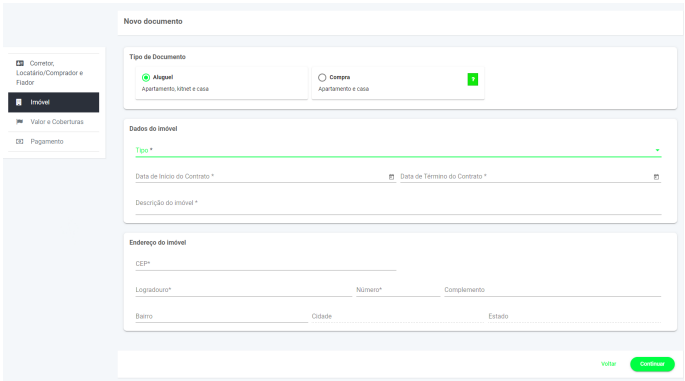
[2] https://angular.io/guide/inputs-outputs e https://angular.io/guide/observables

(do componente “filho”) responsável por monitorar a alteração de dados era acionado. Desse modo, o valor da variável que armazenava o tipo de documento selecionado era transmitido para o componente da tela (que se trata do componente “pai”), o qual também “informava” os demais componentes “filhos” o valor em questão. Por sua vez, os *Observables* tem como função “monitorar” determinados elementos de modo que, se houver alguma mudança de estado, será realizada uma ação. No sistema em questão, o elemento seria a área “Tipo de Documento” e a ação seria acionar a função de atualizar os dados do *Output* e, com isso, as informações contidas nos *Inputs*.

Como é possível observar na Figura 3.2, a opção “Aluguel” é marcada como *default* ao carregar a tela. Com isso, o componente “Dados do imóvel” carrega os dados correspondentes à essa opção. Mais especificamente, há o *drop-down* “Tipo” o qual traz as opções “Apartamento”, “Casa” e “Kitnet” como imóveis disponíveis para alugar. Além disso, os campos das datas se referem às datas de início e término do contrato de aluguel.

Já na Figura 3.3, há o cenário no qual é selecionada a opção de “Compra”. Percebe-se que os campos da aba “Dados do imóvel” foram alterados, de modo que o *drop-down* “Tipo” mostra as opções “Apartamento” e “Casa” como imóveis disponíveis para compra. Os campos de datas também se adequaram à opção selecionada, assumindo os nomes “Data de início da vigência do Documento” e “Prazo final de entrega da escritura”, respectivamente. O campo de descrição do imóvel também foi aumentado para tal cenário, de modo a incentivar o usuário a escrever uma descrição mais completa do imóvel adquirido.

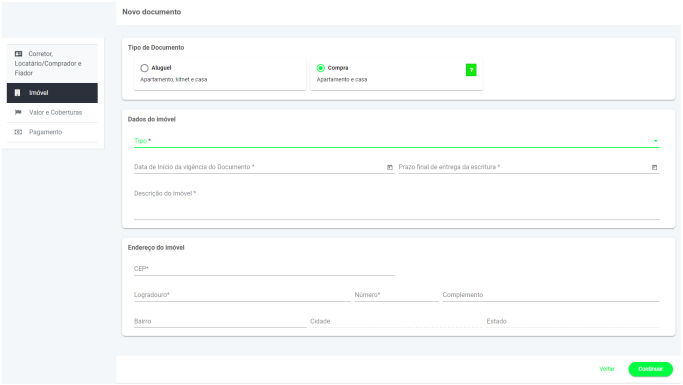
Figura 3.2 – Tela de seleção de tipo de documento - Aluguel



Fonte: Do autor (2022)

Vale ressaltar que, independente da opção selecionada, a aba “Endereço do imóvel” não é alterada, tendo seu preenchimento obrigatório em ambos os cenários. Outro ponto que merece ser mencionado se trata do botão verde com o símbolo de interrogação na Figura 3.3. É possível selecioná-lo não importando a opção de documento e, quando selecionado, abre um *pop-up* explicando os tipos de imóveis disponíveis para compra e aluguel no momento, fornecendo um contato da empresa. Esse botão foi adicionado de modo a esclarecer dúvidas acerca da nova *feature*.

Figura 3.3 – Tela de seleção de tipo de documento - Compra



Fonte: Do autor (2022)

Na parte do *back-end*, foi definido que as regras de negócio já implementadas deveriam sofrer uma revisão, de modo a se adaptarem às novas regras oriundas do novo fluxo. Durante a revisão em questão, foi necessário implementar e alterar testes de unidade de modo a “cobrir” mais cenários. Um exemplo fictício da implementação de tais testes se encontra no Código 2.4 da Seção 2.4.

Além disso, foi fundamental criar e atualizar algumas tabelas no banco de dados do sistema, de modo a armazenarem informações substanciais a serem recebidas e consultadas eventualmente. Dentre tais alterações no banco de dados, tem-se a criação de novas colunas para comportar os dados que, anterior à implementação da *feature* da compra de imóveis, não eram armazenados e/ou mapeados, como o tipo de documento. Para isso, foi feito um ALTER TABLE na tabela “DocumentosGerados” onde a coluna “Tipo” foi adicionada com preenchimento obrigatório e valor *default* 1 (que corresponde a documentos do tipo “aluguel”), como é apresentado no Código 3.1. Vale acrescentar que foi necessário realizar um DROP

DEFAULT para que, ao tentar adicionar um valor na tabela “DocumentosGerados”, se não for passado o dado da coluna “Tipo”, retornar um erro e não preencher como 1.

Código 3.1 – Exemplo de ALTER TABLE em uma tabela utilizando SQL

```
ALTER TABLE DocumentosGerados
ADD Tipo INT NOT NULL DEFAULT 1

ALTER TABLE DocumentosGerados
ALTER Tipo DROP DEFAULT;
```

Fonte: Do autor (2022)

3.1.1 Discussão

Diante dos desafios enfrentados pela estagiária, a implementação do novo fluxo de criação de documentos permitiu que os conceitos abordados, principalmente, durante os estudos da linguagem C# e do *framework* Angular fossem colocados em prática, gerando dúvidas a respeito de tais temas, bem como das regras de negócio em si. Com o auxílio da equipe de desenvolvimento, tais dúvidas foram abordadas e solucionadas, de modo a consolidar o aprendizado das tecnologias utilizadas e do contexto do cliente.

Para o cliente, de maneira resumida, a implementação da nova *feature* era bastante esperada, uma vez que tal fluxo não era automatizado, sendo bastante árduo em termos de custo e de tempo. Assim, além de solucionar os problemas citados anteriormente, a implementação em questão aumentou o escopo de atuação da aplicação utilizada no dia-a-dia dos colaboradores.

3.2 Alterações em *features*

Organizações voltadas para o meio de tecnologia da informação, geralmente, buscam se atualizar com base nas tendências do mercado e nos seus respectivos escopos, de modo a se manterem na “competição” de consumo no meio que atuam. Como afirma Alves (2021),

> “a exigência dos mercados e o próprio reordenamento geo-político do mundo fazem com que as empresas nacionais procurem novas alternativas produtivas, analisando as consequências e otimizando resultados, no sentido de buscar a sobrevivência dentro desse novo quadro que se apresenta.”

Com isso, pode surgir a demanda de realizar alterações nos sistemas computacionais que tais empresas utilizam. Dentre as mudanças que surgiram durante o período de estágio, algumas se destacaram, seja pela complexidade ou pelo objetivo final.

3.2.1 Ajustes em PDFs

Devido ao fato de o sistema principal utilizado pelos clientes emitir documentos de compra e aluguel de imóveis, os documentos em questão deveriam ser disponibilizados em formato PDF e deveriam também permitir sua impressão para futuras assinaturas ou arquivamento. Desse modo, quando ocorriam atualizações nas regras de negócio da empresa que afetavam diretamente cláusulas e campos de contratos, era necessário editar tais documentos de modo a comportarem as novas diretrizes da empresa.

Para realizar tais atualizações, foi utilizado o software de *design* de documentos Crystal Reports[3], o qual foi instalado diretamente na IDE (ambiente de desenvolvimento integrado) Visual Studio.

De modo a explicar o ambiente de desenvolvimento em questão mas manter o sigilo necessário, são utilizados apenas fragmentos de interesse dentro projeto. Assim, como é possível analisar na Figura 3.4, o padrão de organização dos documentos se trata da divisão por seções, as quais podem ou não conter análises lógicas, como o caso do campo “segundoFiador”. Através de uma aba da própria IDE, foi possível adicionar uma expressão que analisava se havia um valor nesse campo. Se o resultado fosse falso, o campo não apareceria no PDF quando o documento fosse gerado.

Figura 3.4 – Abas “Dados do contratante” e “Dados do Fiador” do PDF

Fonte: Do autor (2022)

[3] https://www.sap.com/brazil/products/crystal-visual-studio.html

Outra alteração realizada foi a criação de um rodapé contendo informações sobre a empresa, caso o cliente desejasse entrar em contato. Para tanto, foi necessário apenas selecionar um botão de criação de seção e colocar um campo de texto. Além disso, foi escrito o texto que a empresa desejava, como ilustra a Figura 3.5.

Figura 3.5 – Rodapé do PDF após a alteração feita

Fonte: Do autor (2022)

3.2.2 Bloqueio de campos para edição

Durante o período de estágio, o cliente alterou algumas regras de negócio, o que impactou diretamente no funcionamento dos sistemas da organização em questão. Dentro do projeto *front-end*, foram detectadas algumas abas as quais possuíam campos que, em determinados cenários, deveriam ser bloqueados para edição. Esse bloqueio se mostrou necessário pois, nos cenários específicos, a alteração dos mesmos afetaria diretamente o valor dos documentos gerados, o que acarretaria em inconsistências na contabilidade do cliente.

O código do projeto supracitado já possuía implementadas a análise e alteração de *status* do documento. Desse modo, foi necessário criar uma variável booleana responsável por, nos arquivos HTML das respectivas telas, bloquear os campos a partir da análise já implementada. Um ponto interessante foi o fato de que a lógica já codificada também foi alterada, de modo a comportar mais mudanças e análises similares à demanda em questão. Um exemplo de campo bloqueado trata-se do campo “Nome” na Figura 3.1.

Vale a pena ser mencionado que a implementação do bloqueio de campos foi realizada antes da criação da *feature* explicada na Seção 3.1. Desse modo, a estagiária passou por diversos desafios e aprendizados no que diz respeito, principalmente, à compreensão do código-fonte para desenvolver a alteração de cam-

pos. O que foi aprendido possibilitou o desenvolvimento do novo fluxo de criação de documentos com mais facilidade, ainda que tenham surgido outros desafios.

3.2.3 Remodularização de projeto

No início do estágio, a equipe de desenvolvimento estava realizando uma remodularização no projeto *back-end* de uma aplicação. Vale citar que foi decidido realizar essas mudanças de modo que a arquitetura do projeto fosse mais condizente com a realidade do sistema no momento, além de otimizar e melhorar o respectivo código. Assim, a estagiária atuou com os demais colegas para finalizar essas alterações. Vale acrescentar que, durante tal desenvolvimento, foi definido que os princípios SOLID (introduzidos na Seção 2.6) deveriam ser seguidos.

De maneira resumida, tal remodularização consistia em remover o máximo de funções que utilizavam um sistema legado, cuja arquitetura era hexagonal (ver Seção 2.7), e "reimplementá-las" dentro do novo modelo de arquitetura proposto, que se trata da Arquitetura limpa, ou *Clean architecture* (ver Seção 2.8).

Além da adequação à Arquitetura limpa, também foi feita a implementação de testes de unidade automatizados (ver Seção 2.4). Desse modo, tanto os validadores quanto os próprios *use case*s passam por diversos testes de caso de uso antes de serem colocados em um ambiente de desenvolvimento. Vale acrescentar que foram feitos testes manuais de modo a analisar o comportamento do sistema após as mudanças feitas.

A estagiária realizou, no geral, migração de uma classe simples, uma vez que era necessário que se familiarizasse com o código antes de trabalhar com elementos mais complexos. Uma das classes alteradas foi a busca pela corretora de imóveis a partir do CNPJ. Após a implementação da classe em questão, além dos testes de unidade, foi necessário adequar classes que a utilizavam, de modo a manter o funcionamento correto do código. Um exemplo de tais classes se trata da "InserirAluguelUseCase", que utilizava a busca em questão para validar os dados da corretora que estava realizando o aluguel.

3.2.4 Discussão

Ao iniciar as tarefas de alterações de *features*, a estagiária foi orientada pelo líder da equipe de desenvolvimento a estudar os conceitos que seriam abordados durante sua atuação, bem como em outros tipos

de desenvolvimento dentro do sistema designado para a equipe. Esse estudo inicial permitiu que a estagiária adquirisse conhecimento acerca de temas como Arquitetura de Software, testes de unidade e linguagem HTML, e pudesse aplicá-los.

Além disso, com as alterações feitas nos sistemas web e nos PDFs, foi possível manter os produtos da empresa em questão atualizados, seguindo os protocolos e necessidades definidos pela empresa. Outro ponto que vale citar foi o fato de que, com a remodularização, a equipe de desenvolvimento sentiu um impacto positivo no que diz respeito ao tempo de codificação e análise de código, além da cobertura de casos de teste.

3.3 Correção de *bugs*

Quando se tem uma aplicação computacional, é comum que esporadicamente ocorram erros e comportamentos inesperados por parte do sistema. Tais erros podem exprimir impactos variáveis, indo de um usuário não conseguir assinar um documento dentro do sistema por não estar cadastrado no banco de dados até erros de cálculos de valores no *front-end* devido à erros de lógica. Tomando os cenários presenciados durante o estágio, alguns chamam mais atenção devido ao nível de impacto e às formas de atuação e correção.

3.3.1 Erro ao tentar gerar um documento

Um exemplo trata-se de um *bug* o qual um usuário não conseguia finalizar a criação de um documento. Sempre que tal usuário selecionava o botão para gerar o documento, surgia uma mensagem na tela afirmando que o CNPJ do mesmo era inválido, o que se mostrava inconsistente uma vez que, para realizar o *login* na plataforma em questão, era necessário informar o CNPJ e o usuário estava logado no momento do erro.

Devido ao fato de se tratar de inconsistência de dados, o primeiro passo foi entrar no banco de dados do sistema e analisar se o usuário estava cadastrado na respectiva tabela. Para isso, foi usado a instrução SELECT do SQL para poder obter todos dados, se houvessem, do usuário em questão a partir de seu CNPJ, como mostra o Código 3.2. Com a execução dessa *query*, foi constatado que o usuário em questão não estava cadastrado no banco de dados analisado.

Código 3.2 – Exemplo de SELECT em uma tabela utilizando SQL

```sql
-- Exemplo de SELECT a partir do CNPJ do usuário
SELECT * FROM CORRETORAIMOVEIS
WHERE cnpj = 61767183000160
```

Fonte: Do autor (2022)

Um ponto que vale a pena mencionar é que o sistema em questão consome mais de um banco de dados, de modo que partes diferentes de tal aplicação acessam bancos de dados distintos. Assim, o erro em questão foi provocado pois o banco de dados acessado para validar o *login* não era o mesmo utilizado para "conferir" os dados antes de gerar um documento.

Para corrigir o erro, foi necessário obter os dados do usuário do banco de dados no qual o usuário em questão estava cadastrado e adicionar tais dados no banco de dados utilizado para validar a geração de documentos através de um INSERT, como é possível observar no Código 3.3.

Código 3.3 – Exemplo de INSERT em uma tabela utilizando SQL

```sql
-- Exemplo de INSERT para dados de um usuário na tabela
    'CORRETORAIMOVEIS'
INSERT INTO CORRETORAIMOVEIS (id, cnpj, nome, email, senha, taxa)
VALUES (NEWID(), 61767183000160, 'Corretora Geraldo Fernando Vieira',
    'geraldoFVieiraCorretora@email.com', '[redacted]', 0.015);
```

Fonte: Do autor (2022)

3.3.2 Listagem de documentos atrasados

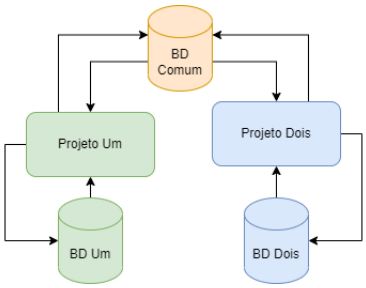
Dentro do sistema, havia uma página na qual eram carregados os documentos emitidos. O incidente em questão se tratou do fato de que os documentos mais recentes não estavam aparecendo nessa tela.

Como se tratava de um erro recorrente, o primeiro "passo" foi entrar em contato com desenvolvedores mais experientes da equipe de desenvolvimento para alinhar os conhecimentos. Após esse alinhamento, entrou-se em contato com a equipe de infraestrutura e solicitou-se o reprocessamento de dados. Após o reprocessamento, os documentos "apareceram" na tela esperada.

A partir de uma análise mais minuciosa, percebeu-se que o *bug* ocorria devido a um *timeout* (ou interrupção) que ocorria no *back-end* do sistema. Vale ser citado que, no sistema em questão, há dois projetos, com seus respectivos bancos de dados, que acessam um terceiro banco, como mostra a Figura 3.6.

A causa raiz do erro se devia ao fato de que, ao tentar salvar dados no banco de dados em comum, o *back-end* de um dos projetos não possuía tratamento de exceções. Assim, se ocorria um cenário não previsto, o sistema parava a execução e era necessário reiniciá-lo.

Figura 3.6 – Organização do sistema



Fonte: Do autor (2022)

Para corrigir essa causa raiz, foi necessário implementar um tratamento de exceção no trecho de código que analisava os dados a serem salvos a partir de um projeto, no banco de dados "comum" do sistema.

3.3.3 Inconsistência de dados

Ao tratar da geração de documentos, um cenário relativamente comum se trata de *bugs* envolvendo inconsistência de dados. Durante o período de estágio, diversos erros desse tipo ocorreram e, em sua maioria, foram corrigidos de maneira similar.

Podem ser citados alguns exemplos: (i) o somatório total do valor das parcelas a serem pagas e somatório do valor total do documento não eram iguais; (ii) um documento possuía todos os dados preenchidos de maneira correta mas ocorria erro ao tentar emiti-lo; e (iii) carregamento incorreto de dados ao abrir um documento criado.

Para atuar em tais incidentes, primeiramente eram feitas consultas nos respectivos bancos de dados de modo a conferir os dados. Ao constatar que os dados estavam coerentes com o cenário esperado, era necessário solicitar ao líder de desenvolvimento da equipe para realizar um reprocessamento manual. Tal reprocessamento, resumidamente, tratava os dados referentes à criação ou consulta feita (dependendo do cenário) de modo a corrigir ou criar tais elementos. Geralmente, os motivos por trás de tais erros eram *timeout* ou o fato dos dados ainda não terem passado na rotina completa para serem acessados pelo usuário.

3.3.4 Discussão

De modo geral, os *bugs* encontrados durante o período de estágio foram essenciais para que a estagiária se familiarizasse com os cenários das aplicações que eram de responsabilidade da equipe de desenvolvimento no qual ela foi designada. Além disso, houve a possibilidade de corrigir erros que afetavam diretamente os sistemas, de modo a gerar um impacto positivo no trabalho dos clientes e o desenvolvimento do senso de "dono"[4].

Um ponto que merece destaque trata-se do motivo o qual ocorriam diversos *timeouts* durante a execução dos sistemas. A razão por trás de tal cenário é o fato de que algumas das aplicações recebem uma quantidade massiva de dados advindas de outros sistemas, de modo que algumas dessas requisições podem conflitar em termos de tempo de execução, levando a *timeouts*. Outro detalhe é o fato de que a conexão entre os sistemas transcorre de maneira distinta, de modo que a rede de uma aplicação pode estar mais "lenta" que a de outra, ocasionando, entre outros aspectos, um *delay* relevante e, consequentemente, um *timeout*.

[4] Trata-se de agir dentro da empresa de maneira responsável e proativa, sempre levando em conta os interesses da mesma. (LEITE, 2021)

4 CONCLUSÃO

O estágio de desenvolvedor *full-stack* trata-se de uma experiência engrandecedora nos mais diversos aspectos. No estágio tratado no documento em questão, foi possível abordar conceitos como os principais tipos de desenvolvimento de software (Seção 2.2), incluindo o próprio conceito de *full-stack*. Outro ponto interessante trata-se do estudo de diferentes arquiteturas de software (Seções 2.7 e 2.8).

Muitas atividades foram realizadas durante o referente estágio, tendo como principais objetivos agregar conhecimento e prover melhorias nos produtos tratados. Um dos desenvolvimentos em questão trata-se da criação de *features* (Seção 3.1). Através desse desenvolvimento, foi possível aprender sobre a regra de negócios do cliente, além do funcionamento do *framework* Angular. Outro ponto interessante foi o fato de que a implementação da "comunicação" entre o *front-end* e o *back-end* da aplicação levou a um conhecimento de técnicas de mapeamento de dados e criação de requisições.

Em relação às alterações de *features* (Seção 3.2), foram abordados diversos escopos de desenvolvimento. Isto é, houve tarefas voltadas para a API e outras implementações voltadas para o *front-end*, além de alterações no banco de dados. Tais atividades permitiram a estagiária vivenciar algumas das diferentes maneiras que um profissional da área de tecnologia pode trabalhar e se especializar. Além disso, houve a oportunidade de compreender o funcionamento completo de uma aplicação, desde a interface do usuário até a consulta ao banco de dados.

No que se trata da correção de *bugs* (Seção 3.3), houve ocasiões nas quais a estagiária precisou entrar em contato direto com usuários do sistema para conferir dados ou mesmo solicitar repetição de cenários de erro. Um exemplo desse tipo de situação foi a necessidade de acionar uma corretora para compreender o passo-a-passo que culminou em um determinado incidente. Com isso, a estagiária pôde reforçar os conhecimentos acerca das regras de negócios do cliente, além de compreender o funcionamento do um sistema legado no qual eram criados os documentos. Vale acrescentar que também foi possível desenvolver mais confiança ao lidar com clientes ou usuários finais das aplicações.

Com isso, percebe-se que no estágio realizado na empresa DTI Digital foi possível aplicar e consolidar conhecimentos adquiridos no decorrer do curso de Bacharelado em Ciência da Computação, tais como

programação orientada a objetos, engenharia de software, organização de projeto, *DevOps*, refatoração e arquitetura de software.

Além disso, tratou-se de uma oportunidade de aprendizado, não apenas da cultura da empresa e do desenvolvimento ágil (principalmente o *Scrum*), mas também da linguagem C#, dos *frameworks* Angular e .NET, além da atuação em casos reais de empresas parceiras.

REFERÊNCIAS

ALVES, F. Competitividade das empresas brasileiras no mercado internacional: Estratégias através de novos indicadores mercadológicos. **VIII Simpósio de Excelência em Gestão e Tecnologia**, 2021.

BACK, J. **Testes de unidade e integração com .NET CORE e xUnit**. 2020. Disponível em: <https://jozimarback.medium.com/testes-de-unidade-e-integracao-com-net-core-e-xunit-fad7c18a29a1>.

BECK, K. et al. **Manifesto para Desenvolvimento Ágil de Software**. 2001. Disponível em: <https://agilemanifesto.org/iso/ptbr/manifesto.html>.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao Teste de Software**. 1. ed. São Paulo: Elsevier, 2007.

FERNANDES, M. **Teste de Unidade e Teste de Integração: O que são?** 2018. Disponível em: <https://medium.com/@mateus1198/teste-de-unidade-e-teste-de-integracao-o-que-sao-de58d7a3d3d2>.

IRIGOYEN, A. et al. **Jornada DevOps: Unindo cultura ágil, *Lean* e tecnologia para entrega de software com qualidade**. 1. ed. Rio de Janeiro: Brasport, 2019.

JUNIOR, A. A. de B. **Modularização**. 2011. Disponível em: <https://antoniojr.webnode.com.br/_files/200000025-6cfe46df80/PRO-Aula12_Modularizacao.pdf>.

LEITE, G. **Senso de dono na empresa: o que é, benefícios e como desenvolver**. 2021. Disponível em: <https://www.feedz.com.br/blog/senso-de-dono-na-empresa/>.

LEUCOTRON. **Empresas que usam Scrum para aumentar o sucesso do negócio.** 2021. Disponível em: <https://blog.leucotron.com.br/afinal-por-que-as-empresas-estao-investindo-na-metodologia-scrum/>.

LIMA, J. A.; BESSA, J. M. de. Utilização do Scrum no desenvolvimento de sistemas computacionais na empresa núcleo da cidade de Ceres-GO. **Simpósio Unificado dos cursos de Sistemas de Informação da Universidade Estadual de Goiás**, 2016.

LUBIENIECKI, A. **O que é e como funciona o Planning Poker?** 2021. Disponível em: <https://www.luby.com.br/o-que-e-e-como-funciona-o-planning-poker/>.

LUCANIA, I. **O que é *DevOps*?** 2019. Disponível em: <https://konia.com.br/o-que-e-devops/>.

MARTIN, R. C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. 1. ed. [S.l.]: Pearson, 2017.

NETO, A. C. D. **Bancos de Dados Relacionais**. 2011. Disponível em: <https://www.devmedia.com.br/bancos-de-dados-relacionais/20401>.

NORTHWOOD, C. **The Full Stack Developer: Your Essential Guide to Everyday Skills Expected of a Modern Full Stack Web Developer**. 1. ed. Manchester: Apress Media LLC, 2018.

PAIXÃO, J. R. da. **O que é SOLID: O guia completo para você entender os 5 princípios da Programação Orientada a Objetos**. 2019. Disponível em: <https://medium.com/desenvolvendo-com-paixao/2b937b3fc530>.

PALMA, A. A. **Setor externo: bons ventos até aqui, mas pode haver turbulências no caminho**. 2021. Disponível em: <https://www.ipea.gov.br/cartadeconjuntura/index.php/tag/balanca-comercial/>.

ROVEDA, U. **O que é deploy, para que serve, vantagens e como fazer deploy**. 2021. Disponível em: <https://kenzie.com.br/blog/o-que-e-deploy/>.

SCHWABER, K.; SUTHERLAND, J. **O Guia do scrum**. 2020. Disponível em: <https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Portuguese-European.pdf>.

SHIOTSU, Y. **A Beginner's Guide to Back-End Development**. 2017. Disponível em: <https://www.upwork.com/resources/beginners-guide-back-end-development\#fundamentals>.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011.

SOUTO, M. O que é front-end e back-end? **Blog Alura - Artigos**, 2019. Disponível em: <https://www.alura.com.br/artigos/o-que-e-front-end-e-back-end>.

TERRA, R.; VALENTE, M. T.; BIGONHA, R. S. An approach for extracting modules from monolithic software architectures. **IX Workshop de Manutenção de Software Moderna**, 2012. Disponível em: <http://professores.dcc.ufla.br/~terra/publications_files/2020_sbes.pdf>.

VALENTE, M. T. **Engenharia de Software Moderna**. 1. ed. [S.l.]: Independente, 2020.

ZARELLI, G. B. **Descomplicando a Clean Architecture**. 2020. Disponível em: <https://medium.com/luizalabs/descomplicando-a-clean-architecture-cf4dfc4a1ac6>.