RENAN REZENDE MODENESE

DESENVOLVIMENTO DE SOFTWARE DE MOBILIDADE URBANA NA BUZU BRASIL

LAVRAS – MG

2023

RENAN REZENDE MODENESE

DESENVOLVIMENTO DE SOFTWARE DE MOBILIDADE URBANA NA BUZU BRASIL

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

Profª. DSc. Renata Lopes Rosa
Orientadora

LAVRAS – MG

2023

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos

da Biblioteca Universitária da UFLA

Modenese, Renan Rezende
Desenvolvimento de software de mobilidade urbana na Buzu Brasil / . 1^a ed. – Lavras : UFLA, 2023.
34 p. : il.

Relatório de estágio (graduação)–Universidade Federal de Lavras, 2023.
Orientadora: Profª. DSc. Renata Lopes Rosa.
Bibliografia.

1. TCC. 2. Monografia. 3. Dissertação. 4. Tese. I. Universidade Federal de Lavras. II. Desenvolvimento de *software* de mobilidade urbana na Buzu Brasil.

RENAN REZENDE MODENESE

DESENVOLVIMENTO DE SOFTWARE DE MOBILIDADE URBANA NA BUZU BRASIL

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

APROVADA em 08 de Fevereiro de 2023.

Prof. DSc. Demóstenes Zegarra Rodriguez	UFLA
Profª. DSc. Renata Teles Moreira	UFLA

Profª. DSc. Renata Lopes Rosa
Orientadora

LAVRAS – MG
2023

AGRADECIMENTOS

Agradeço primeiramente à minha família e amigos, o apoio que tornou essa conquista possível.

À professora Renata Lopes Rosa a orientação.

À Buzu Brasil a oportunidade e aprendizado.

RESUMO

O presente documento trata-se de um relatório de estágio realizado na empresa Buzu Brasil, que abrange o desenvolvimento de funcionalidades para uma API REST responsável pelo *back-end* de uma aplicação móvel. Durante o período de estágio, foram possíveis o aprendizado e prática dos conceitos de Arquitetura Limpa e Padrões de Projeto, que guiaram as atividades desenvolvidas. As tecnologias utilizadas incluem a linguagem TypeScript, o ambiente de desenvolvimento Node.js e o *framework* web Express.js.

Palavras-chave: Desenvolvimento de software. API REST. Arquitetura Limpa. *Desing Patterns*. Áreas de conhecimento.

LISTA DE FIGURAS

Figura 2.1 – Diagrama da Arquitetura Limpa 10
Figura 2.2 – Exemplo de *Factory* 14
Figura 2.3 – Exemplo de *Strategy* 15
Figura 2.4 – Exemplo de *value object* 16
Figura 4.1 – Caso de uso de *NearRouteLineStopPoints* 21
Figura 4.2 – Controlador de *NearRouteLineStopPoints* 22
Figura 4.3 – *Factory* de *NearRouteLineStopPoints* 23
Figura 4.4 – Diagrama UML do caso de uso de verificação 24
Figura 4.5 – Entidade *Organizations* 25
Figura 4.6 – *Mapper* de *Organizations* 26
Figura 4.7 – Interface do repositório de *Organizations* 27
Figura 4.8 – Repositório de *In-memory Organizations* 28
Figura 4.9 – Repositório do ORM de *Organizations* 29
Figura 4.10 – Interface do *AppleProvider* 29
Figura 4.11 – *AppleProvider* 31

SUMÁRIO

1 INTRODUÇÃO . . . 7
1.1 Objetivos . . . 7
1.2 Estrutura do documento . . . 8
2 REFERENCIAL TEÓRICO . . . 9
2.1 REST . . . 9
2.2 Arquitetura de software . . . 9
2.3 Arquitetura Limpa . . . 10
2.3.1 Regra de dependência . . . 11
2.3.2 Princípio da Inversão de Dependência . . . 11
2.3.3 Camadas da Arquitetura Limpa . . . 11
2.3.3.1 Camada de entidades . . . 12
2.3.3.2 Camada de casos de uso . . . 12
2.3.3.3 Camada de adaptadores de interface . . . 12
2.3.3.4 Camada de *frameworks* e *drivers* . . . 13
2.3.3.5 Camada principal e de configuração . . . 13
2.4 *Design Patterns* . . . 13
2.4.1 *Factory* . . . 14
2.4.2 *Strategy* . . . 14
2.4.3 *Value Objects* . . . 15
3 DESCRIÇÃO DA EMPRESA . . . 17
3.1 Estrutura organizacional . . . 17
3.2 Processos . . . 18
3.2.1 Scrum . . . 18
3.2.2 *Code review* . . . 19
3.2.3 *Pair programming* . . . 19
3.3 Tecnologias . . . 19
3.4 Descrição do sistema . . . 20
4 ATIVIDADES REALIZADAS . . . 21
4.1 Caso de uso de obter paradas mais próximas . . . 21
4.2 Refatoração das rotas de verificação e confirmação . . . 23
4.3 Módulo *Organizations* . . . 24

4.4 **Provider de autenticação da Apple** 30
4.5 **Discussão** 32
5 **CONCLUSÃO** 33
REFERÊNCIAS 34

1 INTRODUÇÃO

O objetivo deste documento é descrever algumas das atividades realizadas no decorrer do estágio em desenvolvimento de software na empresa Buzu Brasil, entre 08/08/2022 a 07/12/2022. A Buzu desenvolve um aplicativo na área de mobilidade urbana e tem sua sede localizada em Divinópolis (MG). O aplicativo tem como objetivo ampliar a oferta de transporte público, em especial em cidades do interior, e assim alcançar um mercado ainda inexplorado por soluções de tecnologia.

A empresa possui uma forte cultura de Arquitetura Limpa, o que se refletiu nas atividades realizadas neste relatório. Conforme Martin (2017), a Arquitetura Limpa é um modelo de arquitetura de software que propõe uma hierarquia bem definida de componentes, onde as entidades e os casos de uso tem posição de destaque. Ela favorece o desenvolvimento de sistemas independentes de fatores externos e testáveis. Com a adoção da Arquitetura Limpa, buscou-se reduzir a complexidade do projeto, assim como a mão de obra necessária e os custos envolvidos.

Entre as ferramentas utilizadas destacam-se a linguagem *TypeScript*, o ambiente de desenvolvimento Node.js e o *framework web* Express.js. Os processos utilizados incluem os ritos *Scrum*, *code reviews* e programação em pares, além de testes automatizados e uma *pipeline* de integração e desenvolvimento contínuos.

A equipe de desenvolvimento *back-end*, no qual o estagiário estava alocado, é composta de três desenvolvedores, incluindo o supervisor de estágio. A empresa conta ainda com dois integrantes na equipe de desenvolvimento *mobile*, um *designer*, e um dos sócios atua como *Product Owner* e *Scrum Master*.

Durante o período do estágio, foi realizado o desenvolvimento de *features*, refatoração de componentes existentes e desenvolvimento de testes para a API da empresa. A API é responsável pelo *back-end* de diferentes aplicativos móveis destinados a diferentes públicos, como passageiros e motoristas, sendo portanto de importância central para o negócio.

1.1 Objetivos

O objetivo do estágio foi de contribuir no desenvolvimento do *back-end* de uma aplicação móvel, que será disponibilizada para o público geral ainda no primeiro semestre de 2023. Sendo o primeiro e principal produto da empresa, a entrega da aplicação possui impacto direto no negócio da empresa e sua participação no mercado.

As atividades realizadas durante o estágio incluíram o desenvolvimento de novas funcionalidades, a refatoração de funcionalidades pré-existentes e o desenvolvimento de testes, entre outras. A realização dessas atividades colaborou com o bom andamento do processo de desenvolvimento e com o cumprimento do prazo de entrega da aplicação.

Essas atividades foram fundamentais para a obtenção de habilidades e conhecimentos essenciais para a carreira de desenvolvedor de software. Outros conceitos como qualidade e arquitetura de software também estiveram presentes, e unidos com as atividades práticas, proporcionaram uma compreensão mais ampla do projeto e desenvolvimento de software.

1.2 Estrutura do documento

O presente documento é estruturado de forma a seguir. A Seção 2 é dedicada à revisão dos aspectos teóricos relevantes abordados durante o estágio, fornecendo uma base para o entendimento dos assuntos tratados. Já na Seção 3, é descrita a empresa onde o estágio foi realizado, incluindo sua história, estrutura, processos e tecnologias. A Seção 4 é dedicada à discussão das atividades desenvolvidas durante o estágio, incluindo as tarefas realizadas, os desafios encontrados e as lições aprendidas. Por fim, a Seção 5 apresenta uma conclusão geral do trabalho, incluindo uma reflexão sobre os aspectos mais importantes e as contribuições do estágio para o desenvolvimento profissional.

2 REFERENCIAL TEÓRICO

Este capítulo fornece a fundamentação teórica dos conceitos relacionados às atividades desenvolvidas. É abordado um referencial básico sobre REST e arquitetura de software, bem como os conceitos de Arquitetura Limpa e *design patterns*.

2.1 REST

De acordo com Fielding (2000), o REST é um conjunto de restrições de arquitetura com o objetivo de gerar sistemas distribuídos eficientes, confiáveis e escaláveis.

Conforme Richardson, Amundsen, Ruby (2013), algumas dessas restrições são inerentes ao protocolo HTTP, enquanto outras são obtidas com a correta organização das rotas da aplicação em recursos, além do uso adequado dos verbos HTTP.

Graças a sua simplicidade e flexibilidade, o REST se tornou o padrão para APIs *web* na indústria de software (RED HAT, 2009), sendo uma escolha natural para o *back-end* na Buzu Brasil.

2.2 Arquitetura de software

Segundo Taylor, Medvidovic e Dashofy (2009), a arquitetura de um sistema de software pode ser definida como o conjunto das principais decisões de projeto do sistema. Ela funciona como um desenho que orienta a construção e evolução do software.

Já Bass, Clements e Kazman (2021) definem a arquitetura como uma abstração do sistema, composta por estruturas que facilitem a compreensão do mesmo, onde cada estrutura é um conjunto de componentes correlacionados.

Martin (2017) acrescenta que o objetivo da arquitetura de software é facilitar o desenvolvimento, implantação, operação e manutenção de um sistema, minimizando o tempo de entrega do sistema de maneira sustentável.

A arquitetura de software abrange uma grande variedade de modelos e padrões. Na próxima seção será abordado o modelo da Arquitetura Limpa, publicado inicialmente no blog do autor (MARTIN, 2012), e posteriormente como livro (MARTIN, 2017), sendo hoje amplamente divulgado.

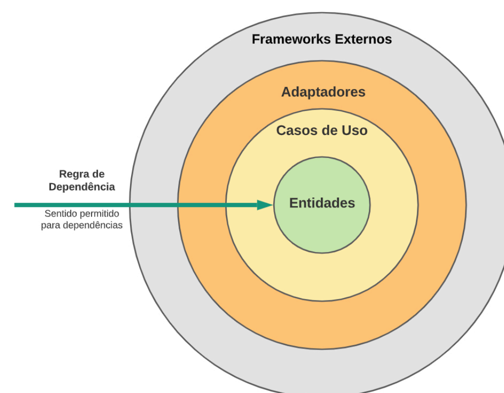
2.3 Arquitetura Limpa

A empresa Buzu possui uma forte cultura de Arquitetura Limpa, a qual atua como norte na estruturação e desenvolvimento do projeto. Dessa forma, se faz necessária a abordagem do conceito.

De acordo com Martin (2017), nas últimas décadas surgiram diversos modelos de arquitetura de software com um objetivo comum, a separação de interesses. Esse objetivo é frequentemente alcançado pela divisão do software em camadas.

Ainda segundo o autor, a Arquitetura Limpa é um modelo que integra conceitos de algumas dessas arquiteturas, e assim como elas divide o software em camadas, conforme ilustrado do diagrama da Figura 2.1. No centro estão as entidades e casos de uso, enquanto nas camadas externas estão adaptadores e a infraestrutura. Quanto mais interna a camada, mais alto o nível de abstração do código contido nela.

Figura 2.1 – Diagrama da Arquitetura Limpa



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

Martin (2017) afirma também que os sistemas produzidos no modelo de Arquitetura Limpa possuem como características a independência de *frameworks*, interface gráfica, banco de dados ou qualquer agente externo. Esse desacoplamento também permite que os casos de uso, que implementam as regras de negócio críticas da aplicação, sejam testados de forma isolada.

No restante da seção, serão discutidos alguns dos conceitos centrais da Arquitetura Limpa, como a regra de dependência e o princípio da inversão de dependência, este apresentado originalmente pelo autor como parte do acrônimo SOLID, que descreve um conjunto de princípios do projeto de software. Também são discutidas as camadas que compõe o modelo da Arquitetura Limpa.

2.3.1 Regra de dependência

Essa regra diz que dependências de software podem apontar somente para dentro, em direção às políticas de alto nível. Isso significa que nada em uma camada interna pode conhecer sobre algo em um camada externa. Se um caso de uso precisa de um serviço ou dependência externa, ele deve chamar uma interface localizada na mesma camada, e um adaptador na camada externa implementa essa interface.

Segundo Martin (2017, p. 206)

> pelo mesmo princípio, formatos de dados usados em um círculo externo não deveriam ser usados em camadas internas, especialmente se esses formatos são gerados por um *framework*. Nós não queremos que nada em uma camada externa impacte as camadas internas.

2.3.2 Princípio da Inversão de Dependência

Esse princípio diz que sempre devemos depender de abstrações, nunca de implementações concretas.

Segundo Martin (2017, p. 80) "o código que implementa políticas de alto nível não deveria depender de código que implementa detalhes de baixo nível. Em vez disso, os detalhes devem depender das políticas." Essa inversão de dependência, onde os detalhes dependem das políticas, é possível com o uso adequado de interfaces e *providers*, possibilitando o desacoplamento do código.

O autor ainda demonstra que interfaces são menos voláteis do que implementações, concluindo que "arquiteturas de software estáveis são aquelas que evitam depender de concreções voláteis, e favorecem o uso de interfaces abstratas."

Para seguir essa regra, a criação de objeto concretos voláteis exige manipulação especial, como o uso de *Factories* ou *Abstract Factories* que conectam os casos de uso as implementações concretas.

2.3.3 Camadas da Arquitetura Limpa

De acordo com o Martin (2017), o número de camadas da Arquitetura Limpa não é fixo, variando conforme a necessidade específica de um sistema. Para isso devemos apenas observar a regra de dependência, de forma que as dependências apontem sempre para o centro do modelo.

2.3.3.1 Camada de entidades

Entidades representam conceitos-chave do domínio do problema, encapsulando dados e regras de negócio críticos da aplicação.

Segundo Lemos (2022) as entidades são projetadas para serem livres de dependências externas e tão independentes do mecanismo de persistência quanto possível. Dessa forma, elas são pouco propensas a mudanças por fatores externos.

Outro benefício da camada de Entidade é que ela permite um alto grau de flexibilidade e reutilização.

2.3.3.2 Camada de casos de uso

De acordo com Martin (2017, p. 206), o *software* nessa camada "contém regras de negócio específicas da aplicação. Ela encapsula e implementa todos os casos de uso do sistema. Esses casos de uso orquestram o fluxo de dados de e para as entidades".

Ainda segundo o autor, mudanças nessa camada não devem afetar as entidades. Da mesma forma, essa camada deve estar protegida de mudanças das camadas externas, como a interface do usuário ou um *framework*. A camada de casos de uso deve ser isolada de preocupações externas.

2.3.3.3 Camada de adaptadores de interface

Nas palavras de Martin (2017, p. 207), essa camada é composta de "um conjunto de adaptadores que convertem os dados para o formato mais conveniente para os casos de uso e entidades, e para o formato mais conveniente para algum agente externo como o banco de dados ou a *web*". Isso inclui os controladores, *middlewares* e *mappers*.

Ainda segundo o autor, "de maneira similar, dados são convertidos, nessa camada, do formato mais conveniente para entidades e casos de uso, para o formato mais conveniente para o banco de dados ou outras formas de persistência".

Lemos (2022, p. 26) reforça que "se existir a comunicação do sistema com quaisquer outros agentes externos, haverá um adaptador nessa camada para mediar a comunicação entre o sistema e o agente".

2.3.3.4 Camada de *frameworks* e *drivers*

Conforme Martin (2017, p. 207), a camada externa "é geralmente composta de *frameworks* e ferramentas como banco de dados e um *framework web*. Nós mantemos essas coisas do lado de fora onde elas podem causar pouco dano a aplicação".

Nesta camada são implementados adaptadores que se comunicam diretamente com código externo à aplicação, incluindo implementações concretas de repositórios e serviços como criptografia e gerenciador de *tokens*.

2.3.3.5 Camada principal e de configuração

De acordo com Lemos (2022, p. 26), "essa camada é que faz todas as conexões entre as interfaces contidas nas camadas mais internas da arquitetura com os adaptadores e as implementações concretas nas camadas mais externas, configurando o sistema para sua execução e executando-o".

Factories, configurações de rotas, a aplicação principal e o servidor web fazem parte dessa camada.

Em resumo, a adoção de Arquitetura Limpa no desenvolvimento de software oferece vários benefícios, incluindo maior capacidade de manutenção, escalabilidade e flexibilidade. Além disso, a Arquitetura Limpa promove a criação de código reutilizável e testável, o que pode levar a economia de tempo e custos a longo prazo.

2.4 *Design Patterns*

Outro conceito recorrente nas atividades desenvolvidas, os *design patterns* são padrões úteis no desenvolvimento e manutenção de software orientado a objeto, introduzidos pela influente obra de Gama et al (1995). Aqui são discutidos apenas os padrões utilizados nas atividades do estágio, que representam dois dos grupos de padrões apresentados pelos autores: padrões criacionais e comportamentais. Também será abordado um padrão de projeto retirado do paradigma *Domain-driven design* (GAMMA et al., 1995).

2.4.1 *Factory*

Um padrão criacional, o *factory method* é simplesmente um método que faz a criação de um objeto e suas dependências, retornando uma instância desse objeto. Esse padrão é frequentemente usado em conjunto com a técnica de injeção de dependências.

Segundo Gamma et al. (1995, p. 109)

> o uso desse padrão elimina a necessidade de acoplar implementações concretas no código. O código pode depender apenas de interfaces abstratas, e será compatível com qualquer implementação concreta dessas interfaces.

Outra vantagem desse método *factory* é permitir a criação dinâmica de objetos em tempo de execução, possibilitando a criação de objetos de classes diferentes conforme necessário.

A Figura 2.2 ilustra o código de um *factory method*. No exemplo podemos observar que o construtor da classe é privado, sendo o *factory* o único meio de criar novas instâncias. O método *factory* é estático, ou seja, pode ser chamado diretamente da classe, sem a necessidade de um objeto.

Figura 2.2 – Exemplo de *Factory*

```
class Entity {

    private constructor() {}

    public static create() {
        return new Entity()
    }
}
```

Fonte: Do autor (2023)

O *factory* é provavelmente o padrão de projeto mais utilizado nas atividades desenvolvidas, sendo sua flexibilidade e utilidade essenciais para a arquitetura do sistema.

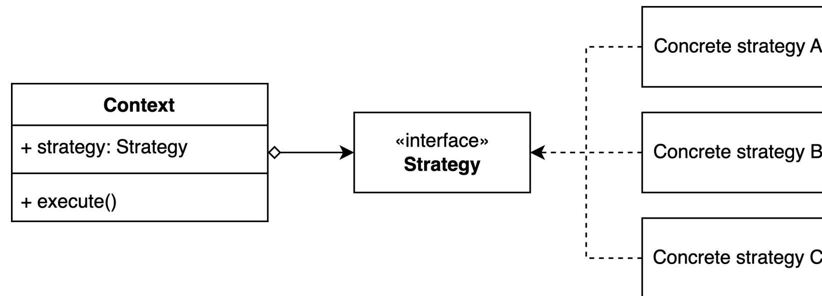
2.4.2 *Strategy*

Conforme Gamma et al. (1995, p. 315), o padrão comportamental *strategy* "define uma família de algoritmos, encapsula cada um deles, e os torna intercambiáveis." Esse padrão permite que diferentes fluxos sejam seguidos, onde o contexto direciona o fluxo para a estratégia adequada.

A Figura 2.3 apresenta um diagrama do padrão *strategy*. A classe *Context* possui como atributo a interface *strategy*, que é implementada por diversas classes concretas. Geralmente

o contexto recebe como entrada um parâmetro que define a estratégia a ser utilizada, sendo responsável apenas pela chamada da estratégia, delegando o trabalho a mesma.

Figura 2.3 – Exemplo de *Strategy*



Fonte: Do autor (2023)

Ainda segundo Gamma et al. (1995), aplicações desse padrão vão desde permitir o uso de variações de um algoritmo, até possibilitar o agrupamento de classes relacionadas, mas com comportamentos distintos. Esse padrão também é notório por reduzir o uso de expressões condicionais.

Uma variação desse padrão é conhecida como *provider*, que também aparece nas atividades desenvolvidas. Introduzido pela Microsoft (2006) como parte do framework .NET, sua similaridade com o padrão *strategy* é discutida (ETHEREDGE, 2009).

O padrão *provider* é especialmente útil para desacoplar implementações concretas dos casos de uso, permitindo que a implementação de um serviço seja facilmente substituída por outra sem que seja necessário alterar o caso de uso, que conhece apenas sua interface.

Como principal diferença, no *strategy* a escolha da estratégia é realizada em tempo de execução, frequentemente delegada ao cliente, enquanto no *provider* a escolha da implementação geralmente é feita em tempo de compilação.

2.4.3 *Value Objects*

Segundo Evans (2003), *value objects* são objetos que representam um único valor ou atributo. A identidade de um *value object* depende apenas do seu valor, e não de um identificador único como as Entidades. Eles permitem uma representação mais precisa do modelo de domínio, tornando mais fácil entender e raciocinar sobre o espaço do problema.

Os *value objects* são usados para encapsular a lógica específica do domínio e para garantir consistência no modelo de domínio. Esses objetos são imutáveis, o que significa que,

uma vez criados, seus valores não podem ser alterados. Eles também ajudam a garantir que os dados sejam válidos, impondo restrições aos valores que contêm.

Vernon (2013), observa que esse padrão foi introduzido pelo paradigma de *Domain-Driven Design*, e é um dos seus blocos de construção fundamentais.

A Figura 2.4 ilustra o código de um *value object*. O método *isValid* delega ao *validator* correspondente a validação dos dados, e será chamado pelo construtor da classe pai. O *value object* possui também um *factory method create* que instancia o objeto a partir do dado primitivos, nesse caso uma *string*. Como a validação dos dados é feita na instanciação dos objetos, existe a garantia da validade dos dados no sistema.

Figura 2.4 – Exemplo de *value object*

```
export class Token extends ValueObject<string> {

  public isValid(value: string): boolean {
    return isTokenValid(value)
  }

  public static create(value: string): Either<
      InvalidTokenError, Token> {
    try {
      const vo = new Token(value)
      return right(vo)
    } catch (err) {
      return left(new InvalidTokenError(value))
    }
  }
}
```

Fonte: Do autor (2023)

No capítulo seguinte, será apresentada a empresa onde aconteceu o estágio, incluindo sua estrutura, processos adotados e tecnologias utilizadas.

3 DESCRIÇÃO DA EMPRESA

Fundada no inicio de 2022 pelos sócios Vinicius Pimenta e Gabriel Mesquita, a Buzu Brasil Tecnologia Ltda. é sediada em Divinópolis (MG), com atuação em todo o estado e regime de trabalho totalmente remoto.

A empresa atualmente desenvolve um produto da área de mobilidade urbana que possui como funcionalidade principal a mediação entre veículos de transporte coletivo e passageiros, com versões diferentes do aplicativo para os diferentes tipos de usuário.

O produto tem objetivo de organizar a mão de obra de motoristas de van, de forma que circulem em rotas e horários pré-definidos, e possam atender a demanda por transporte coletivo, que frequentemente não é atendida de forma satisfatória pelas empresas de ônibus intermunicipais.

As funcionalidades do aplicativo do passageiro incluem a visualização dos veículos no mapa, com informações sobre o trajeto e tempo de chegada, além do controle do pagamento, onde o usuário adiciona saldo e faz o pagamento pelo próprio aplicativo. Já os motoristas podem facilmente compartilhar veículos e rotas, além de manejar e sacar os valores obtidos com as viagens. Como forma de monetização, a empresa receberia com uma pequena taxa sobre cada passagem paga pelo aplicativo.

O produto se encontra em fase final de desenvolvimento, com previsão para publicação ainda no início de 2023.

3.1 Estrutura organizacional

A empresa é composta pelos sócios, que possuem sólidos conhecimentos em desenvolvimento e atuam como líderes técnicos, além de desempenharem funções como *Product Owner* e *Scrum Master*. A empresa conta também com outros dois desenvolvedores *back-end*, dois desenvolvedores *mobile* e um *designer*. O estagiário está alocado na equipe de desenvolvimento *back-end*, trabalhando na API do sistema.

A proximidade dos fundadores com o desenvolvimento possibilitou que a empresa estabelecesse uma forte cultura de Arquitetura Limpa, além de uma sólida infraestrutura de tecnologia, incluindo ampla cobertura de testes, além de integração e desenvolvimento contínuos.

A empresa opera em regime totalmente remoto, utilizando como principal ferramenta de comunicação o *Discord*. A comunicação da empresa é bastante direta, com o uso canais de

áudio para facilitar a interação entre os membros da equipe. O ambiente da empresa é informal e amigável, favorecendo o aprendizado e desenvolvimento profissional.

3.2 Processos

Nesta seção são descritos os processos adotados pela empresa. As demandas do produto, trazidas pelos sócios ou originadas da percepção da própria equipe de desenvolvimento, são organizadas e priorizadas pelo *Product Owner*. As tarefas são discutidas e distribuídas nas reuniões de planejamento, e executadas durante o período de uma *Sprint*. Ao final do desenvolvimento, o código é submetido a um repositório central e revisado pelos demais membros da equipe. Depois de realizadas as alterações necessárias e obtida a aprovação dos membros, o código é incorporado ao repositório.

3.2.1 Scrum

De acordo com o Guia do Scrum (2010), o Scrum é uma metodologia ágil desenvolvida para organizar projetos e fornecer transparência ao processo de desenvolvimento. Ele é baseado em ciclos chamados *Sprints*, além de reuniões periódicas e diferentes funções, como o *Scrum Master* e o *Product Owner*.

Ainda de acordo com o Guia do Scrum (2010), o *Product Owner* é o responsável por gerenciar o *backlog* do produto, com o objetivo de maximizar o valor gerado pela equipe de desenvolvimento. Já o *Scrum Master* é o responsável pela aplicação correta dos ritos Scrum, também com o objetivo de maximizar o valor gerado. Em nossa experiência na Buzu, não identificamos nenhum prejuízo pela centralização desses papéis em uma única pessoa.

A empresa adota o *framework* Scrum como base de seus processos, organizando o trabalho em *Sprints* quinzenais, e praticando ritos como reuniões diárias e reuniões quinzenais de planejamento, onde ocorre também o Scrum *poker* para estimar o tempo necessário nas tarefas, além da retrospectiva da *Sprint* anterior.

Antes de cada Sprint, o *Product Owner* realiza uma organização prévia das tarefas do backlog, adicionando novas demandas e organizando as tarefas de acordo com seu nível de prioridade. Essas demandas podem ser originadas no *design*, que determina quais e como serão as telas do aplicativo, da equipe *mobile*, que pode identificar funcionalidades faltando ou que

precisem de ajustes durante a integração, ou ainda dos sócios da empresa, que trazem novas demandas conforme o avanço do desenvolvimento do produto.

Na reunião de planejamento as tarefas são apresentadas e atribuídas aos desenvolvedores, que estimam o tempo necessário para executar cada uma delas. Esse processo é repetido até que o tempo previsto para aquela *Sprint* seja totalmente alocado.

Todo o processo de desenvolvimento é documentado no *Github*, desde a organização das tarefas do *backlog*, até a submissão e revisão do código. Também são utilizadas planilhas para o controle de informações como a relação entre o tempo gasto e o estimado por tarefa.

3.2.2 *Code review*

Toda nova funcionalidade ou refatoração realizada no sistema é submetida a um repositório central na plataforma *Github*, e deve ser aprovada por todo os membros de uma equipe, de forma a garantir a qualidade do código e evitar a inclusão de erros ou *code smells* no código.

Novamente a ferramenta utilizada é o *Github*, que oferece recursos como revisão, comentários e bloqueio do *pull request* em determinadas condições, ferramentas úteis para controle e gerenciamento da evolução do sistema.

3.2.3 *Pair programming*

Sempre que necessário é feita a programação em pares com um desenvolvedor mais experiente, favorecendo o entendimento do sistema existente e o aprendizado dos aspectos envolvidos. Para isso é utilizado o *Discord*, que permite o compartilhamento de tela e comunicação em canais de áudio.

A comunicação verbal é bastante favorecida na empresa, recorrendo sempre que necessário a chamadas de voz. Pelo nível de maturidade do produto e tamanho da equipe, geralmente o desenvolvimento de documentação não é prioridade.

3.3 Tecnologias

As tecnologias utilizadas pela empresa no *backend* são a linguagem *TypeScript*, o ambiente *Node*.js e o *framework Express*.js.

Como mencionado anteriormente, toda *feature* deve incluir testes de unidade, e quando necessário também testes de integração, o que tem como resultado uma cobertura praticamente total de testes. Os testes são implementados com auxílio das bibliotecas *Jest* e *Supertest*.

Também é utilizado um *pipeline* de integração e desenvolvimento contínuos, que verifica aspectos como formatação, além de executar o conjunto de testes, impedindo que código defeituoso ou fora do padrão seja incluso no sistema. Esse *pipeline* utiliza tecnologias como *ESLint* para garantir uma formatação padronizada do código, *Husky* para realizar verificações antes de um *commit*, *GitHub Actions* na automação dos testes e outras validações realizadas após o *commit*, e finalmente *SonarCloud*, que verifica a existência de *bugs* e *code smells*.

Na próxima seção, serão discutidas algumas das atividades realizadas, incluindo exemplos que abrangem diversas camadas da arquitetura e permitem um melhor entendimento do sistema.

3.4 Descrição do sistema

O sistema é dividido entre o *backend*, que consiste em uma API REST, e as aplicações *mobile*, que são específicas para cada tipo de usuário: passageiro, motorista e colaborador. Todas as aplicações*mobile* utilizam a mesma API no *backend*.

A API do sistema é uma aplicação monolítica estruturada em módulos que dividem o código de acordo com o ator, como no caso dos módulos *Drivers* e *Passengers*, ou pela entidade principal, como no módulo *RouteLines* e *Organizations*. Cada módulo é dividido internamente em camadas como domínio, aplicação e infraestrutura.

A camada de domínio abrange as entidades e casos de uso de um módulo, enquanto a camada de aplicação contém os controladores e rotas, e a camada de infraestrutura reúne os repositórios. Componentes comuns a todos os módulos, como adaptadores, *middlewares*, *providers* e *value objects* em uma espécie de módulo externo, chamado de *shared*.

Toda funcionalidade adicionada ou refatoração realizada deve incluir também testes de unidade e de integração, que são responsabilidade dos desenvolvedores e necessários para aceitação do código submetido.

4 ATIVIDADES REALIZADAS

Neste capítulo são discutidas as principais atividades desenvolvidas no estágio. As atividades foram realizadas no período de 08/08/2022 até 07/12/2022, e incluem o desenvolvimento de novas funcionalidades e refatoração de componentes já existentes.

Essas atividades foram selecionadas de forma a abranger tanto quanto possível do sistema e seus diversos componentes, ao mesmo tempo que evita repetir componentes semelhantes.

4.1 Caso de uso de obter paradas mais próximas

Este caso de uso permite que o usuário, ao enviar sua localização através do aplicativo móvel, receba uma lista dos três pontos de ônibus mais próximos, assim como informações das linhas que passam por aqueles pontos.

Figura 4.1 – Caso de uso de *NearRouteLineStopPoints*

```
export class NearRouteStopPointsUseCase {

  constructor(private routeStopPointsRepository:
      RouteStopPointsRepositoryInterface) {}

  async execute(input: Input): Promise<Either<Output.
      Error, Output.Success>> {
    const routeStopPoints = await this.
        routeStopPointsRepository.findMany()

    const stopPointsWithDistance: StopPointWithDistance
        [] = []
    for (const routeStopPoint of routeStopPoints) {
      stopPointsWithDistance.push({
        distance: this.calculateEuclideanDistance(input.
            passengerLocation, routeStopPoint.point),
        stopPoint: routeStopPoint,
      })
    }

    const nearestStopPoints = this.findNearestStopPoints
        (stopPointsWithDistance)

    const nearestStopPointsPresentation =
        nearestStopPoints.map((stopPoint) =>
      RouteStopPointMapper.toPresentation(stopPoint)
    )

    return right({ nearestStopPoints:
        nearestStopPointsPresentation })
  }
```

Fonte: Do autor (2023)

A Figura 4.1 ilustra o código do caso de uso *NearRouteLineStopPoints*. Ele recebe do controlador as coordenadas da localização do usuário, já validadas e na forma de um *value*

object. O caso de uso é responsável por obter do banco de dados uma lista completa dos pontos de parada, através do repositório correspondente. Em seguida, calcula a distância entre o usuário e cada um desses pontos. Utilizando o método *findNearestStopPoints*, ele ordena os pontos em ordem crescente de distância e seleciona os três mais próximos. Por fim, retorna essas informações formatadas para o cliente com ajuda de um *mapper*.

O controlador, cujo código é ilustrado na Figura 4.2, é responsável por tratar a requisição recebida do cliente. Ele recebe as coordenadas de latitude e longitude passadas como parâmetros da *query string*, converte os valores para números de ponto flutuante, e em seguida instancia um *value object Point* a partir deles. Caso ou valores não tenham o formato esperado, um erro é retornado. Finalmente o *value object* é passado para o caso de uso, e o resultado retornado por ele é repassado ao cliente, podendo ser uma lista com os pontos mais próximos ou um erro.

Figura 4.2 – Controlador de *NearRouteLineStopPoints*

```
type Model = Output.Error | Output.Success

export class NearRouteLineStopPointsController
    implements Controller {

  constructor(private nearRouteLineStopPointsUseCase:
      NearRouteLineStopPointsUseCase) {}

  async handle(httpRequest: HttpRequest): Promise<
      HttpResponse<Model>> {
    const { latitude, longitude } = httpRequest.query)

    if (Number.isNaN(latitude)) {
      return unprocessableEntity(new
          InvalidGeographicCoordinateError(latitude))
    }

    if (Number.isNaN(longitude)) {
      return unprocessableEntity(new
          InvalidGeographicCoordinateError(longitude))
    }

    const passengerLocationOrError = Point.create({
        latitude, longitude })
    if (passengerLocationOrError.isLeft())
      return unprocessableEntity(
          passengerLocationOrError.value)

    const result = await this.
        nearRouteLineStopPointsUseCase.execute({
      passengerLocation: passengerLocationOrError.value,
    })

    if (result.isLeft()) return badRequest(result.value)

    return ok(result.value)
  }
}
```

Fonte: Do autor (2023)

O *factory*, ilustrado na Figura 4.3, é responsável por por criar uma instância de controlador e suas dependências, incluindo o caso de uso e o repositório de pontos de parada. Ele é chamado diretamente pelo *router* do *framework Express*, quando acionada a rota correspondente, gerando as instancias em tempo de execução.

Figura 4.3 – *Factory* de *NearRouteLineStopPoints*

```
export function makeNearRouteLineStopPointsController():
     Controller {

  const routeStopPointsRepository = new
      PrismaRouteStopPointsRepository()

  const nearRouteLinesStopPointsUseCase = new
      NearRouteLineStopPointsUseCase(
    routeStopPointsRepository
  )

  return new NearRouteLineStopPointsController(
    nearRouteLinesStopPointsUseCase
  )
}
```

Fonte: Do autor (2023)

Nessa atividade foi possível implementar toda a funcionalidade de uma rota da API, desde o caso de uso até a *factory* que é chamada na resposta de uma requisição. Com isso foi possível melhorar a compreensão do seu funcionamento.

Esta atividade também incluiu o desenvolvimento de testes, cujo código foi omitido por limitações de espaço. Os testes de unidade validam as regras de negócio do caso de uso, incluindo casos de sucesso, persistência e os possíveis casos de erro. Neles são testados os casos de uso de forma isolados, com ajuda dos repositórios em memória. Já os testes de integração fazem a chamada na API e verificam o retorno obtido em alguns possíveis cenários, e utilizam a infraestrutura real da aplicação, incluindo desde os *middlewares* até o banco de dados.

4.2 Refatoração das rotas de verificação e confirmação

De acordo com Fowler (2018), refatoração é toda mudança feita no software para facilitar seu entendimento e manutenção sem que o comportamento observável seja alterado. Ela ajuda a identificar e corrigir problemas de estrutura e design, tornando o código mais fácil de ler, entender e manter.

Durante o período do estágio, foram realizadas refatorações constantes no projeto para atender a mudanças no design ou necessidades surgidas da integração com a aplicação móvel.

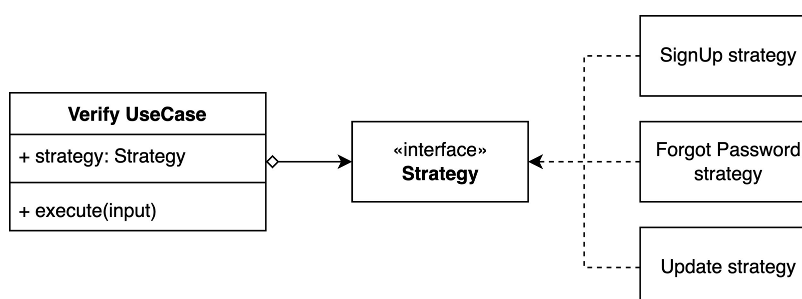
Isso confirma a afirmação de Martin (2017), de que a natureza do *software* é ser mutável, e facilmente modificado.

As rotas de verificação e confirmação estão presentes tanto no módulo do motorista quanto do passageiro, em diferentes fluxos como o cadastro, a recuperação de senha e a atualização do perfil do usuário. A verificação é responsável pela geração de um código de confirmação, que é enviado por e-mail ou telefone, enquanto a confirmação valida esse código e fornece um *token* utilizado na etapa seguinte.

Nessa refatoração foi aplicado o padrão de projeto *strategy*, unificando os diferentes casos de uso. Isso permitiu reduzir o número de rotas e casos de uso de verificação e confirmação, que antes eram específicos para cada objetivo, e passaram a ser estratégias de um único caso de uso.

Como ilustrado na Figura 4.4, o caso de uso possui como propriedade a interface *Strategy*, que é implementada pelas estratégias concretas *SignUp*, *ForgotPassword* e *Update*. A estratégia é selecionada de acordo com o parâmetro *objectiveType*, recebido por ambas as rotas de verificação e confirmação.

Figura 4.4 – Diagrama UML do caso de uso de verificação



Fonte: Do autor (2023)

A refatoração foi bem sucedida e a aplicação do *design pattern strategy* foi útil em outros casos de uso, implementados posteriormente. Em seguida, discutiremos uma funcionalidade que inclui camadas ainda mais internas da arquitetura.

4.3 Módulo *Organizations*

O módulo *Organizations* reúne as funcionalidades referentes as Organizações, grupos de motoristas que compartilham veículos e rotas. O código desenvolvido nessa funcionalidade

Figura 4.5 – Entidade *Organizations*

```
export class Organization extends Entity<OrganizationProps> {

  private constructor({ props, id }: EntityConstructor<OrganizationProps>) {
    super({ props, id })
  }

  get name() {
    return this.props.name.value
  }

  get createdAt() {
    return this.props.createdAt.value
  }

  get updatedAt() {
    return this.props.updatedAt.value
  }

  updated() {
    this.props.updatedAt = UpdatedAt.create(new Date()).value as UpdatedAt
  }

  public static create({
    props,
    id,
  }: EntityConstructor<OrganizationCreateProps>): Either<OrganizationCreateError, Organization> {
    let uuidOrError
    if (id) {
      uuidOrError = Uuid.create(id)
      if (uuidOrError.isLeft()) return left(uuidOrError.value)
    }

    const nameOrError = Name.create(props.name)
    const createdAtOrError = CreatedAt.create(props.createdAt)
    const updatedAtOrError = UpdatedAt.create(props.updatedAt)

    if (nameOrError.isLeft()) return left(nameOrError.value)
    if (createdAtOrError.isLeft()) return left(createdAtOrError.value)
    if (updatedAtOrError.isLeft()) return left(updatedAtOrError.value)

    const organization = new Organization({
      props: {
        name: nameOrError.value,
        createdAt: createdAtOrError.value,
        updatedAt: updatedAtOrError.value,
      },
      id: uuidOrError?.value ?? null,
    })

    return right(organization)
  }
}
```

Fonte: Do autor (2023)

inclui a entidade *Organization*, um *mapper*, a interface do repositório e suas implementações concretas.

O código da entidade *Organization* é exibido na Figura 4.5. Uma entidade é uma representação do domínio do problema, o objeto que será manipulado pelos casos de uso para atender as regras de negócio. Ela possui atributos relativos as organizações, seus métodos *getters* e um

factory method create. Este último é responsável pela inicialização dos *value objects*, e por fim da própria entidade.

O método *create* pode ser usado tanto na criação de uma nova entidade, quanto na criação de uma instância de uma entidade pré-existente. No primeiro caso, nenhum *id* é recebido, e o construtor de *Entity* será responsável por gerar um novo *id*.

A Figura 4.6 ilustra o código do mapper de *Organizations*. O método *toModel* é responsável de converter uma entidade para o formato do banco de dados, enquanto o método *toDomain* realiza a conversão oposta, instanciando uma entidade a partir de um registro do banco de dados.

Figura 4.6 – *Mapper* de *Organizations*

```
export class OrganizationMapper {

  static toModel(organizationEntity: OrganizationEntity): FullOrganizationModel {
    return {
      id: organizationEntity.id,
      name: organizationEntity.name,
      createdAt: organizationEntity.createdAt,
      updatedAt: organizationEntity.updatedAt,
    }
  }

  static toDomain(organizationModel: FullOrganizationModel): OrganizationEntity {
    const {
      id,
      name,
      createdAt,
      updatedAt,
    } = organizationModel

    const organizationOrError = OrganizationEntity.create({
      props: {
        name,
        createdAt,
        updatedAt,
      },
      id,
    })

    if (organizationOrError.isLeft()) throw organizationOrError.value

    return organizationOrError.value
  }
}
```

Fonte: Do autor (2023)

Na camada de infraestrutura, temos a interface do repositório e suas implementações concretas. A interface do repositório é exemplificada na Figura 4.7. Ela define os métodos disponíveis no repositório, bem como quais parâmetros serão aceitos pelos métodos *findFirst* e *exists*.

Figura 4.7 – Interface do repositório de *Organizations*

```
export type FindOrganizationProps = {
  id?: Uuid
  name?: Name
}

export interface OrganizationsRepositoryInterface {
  save(organizationEntity: Organization): Promise<void>
  findFirst(input: FindOrganizationProps): Promise<Organization | null>
  exists(input: FindOrganizationProps): Promise<boolean>
}
```

Fonte: Do autor (2023)

O método *save* é responsável por persistir uma entidade. O método *findFirst* retorna uma organização a partir do seu *id* ou nome. Já o método *exists* apenas nos informa se aquele registro existe, a partir dos mesmos parâmetros.

A Figura 4.8 ilustra a implementação *in-memory* do repositório de organizações. Essa implementação é utilizada nos testes de unidade, onde não temos interesse em testar uma implementação específica de banco de dados, mas sim os casos de uso e regras de negócio.

Já a Figura 4.9 ilustra a implementação concreta do *Object Relational Mapper*, nesse caso especificamente o *Prisma*.

Podemos observar que o método *save* é responsável por receber uma entidade, convertê-la para o formato do banco de dados e persisti-la. Para isso ele utiliza o método *upsert* da *API* do *Prisma*, que pode tanto criar um novo registro como atualizar um pré-existente.

O método *findFirst* apenas chama o método de mesmo nome do *Prisma*, repassando os parâmetros conforme necessário e convertendo o retorno de *undefined* para *null* em caso de nenhum registro ser localizado.

De forma similar, o método *exists* utiliza o mesmo método *findFirst*, mas retornando um valor booleano indicando se o registro existe no banco de dados.

É importante ressaltar que todos os métodos deste repositório são assíncronos, permitindo que as operações do banco de dados sejam realizadas sem bloquear a *thread* principal do servidor.

Essa atividade foi provavelmente a mais relevante entre as desenvolvidas, e a entidade *Organizations* serviu como base para diversos outros casos de uso, tendo sido ela mesma refatorada para incluir novas funcionalidades e relações com outras entidades.

Figura 4.8 – Repositório de *In-memory Organizations*

```
export class InMemoryOrganizationsRepository implements
    OrganizationsRepositoryInterface {

  private organizations: Organization[] = []

  async save(organizationEntity: Organization): Promise<
      void> {
    const itemIndex = this.organizations.findIndex((item
        ) => item.id === organizationEntity.id)

    if (itemIndex === -1) {
      this.organizations.push(organizationEntity)
    } else {
      this.organizations[itemIndex] = organizationEntity
    }
  }

  async findFirst(input: FindOrganizationProps): Promise
      <Organization | null> {
    const organization = this.organizations.find(
      (organization) =>
        (input.id && organization.id === input.id.value)
            ||
        (input.name && organization.name === input.name.
            value)
    )

    return organization ?? null
  }

  async exists(input: FindOrganizationProps): Promise<
      boolean> {
    const exists = this.organizations.find(
      (organization) =>
        (input.id && organization.id === input.id.value)
            ||
        (input.name && organization.name === input.name.
            value)
    )

    if (exists) return true

    return false
  }
}
```

Fonte: Do autor (2023)

Figura 4.9 – Repositório do ORM de *Organizations*

```
export class PrismaOrganizationsRepository implements
    OrganizationsRepositoryInterface {

  async save(organizationEntity: Organization): Promise<
      void> {
    const organizationModel = OrganizationMapper.toModel
        (organizationEntity)

    await prisma.organizationModel.upsert({
      create: organizationModel
      update: organizationModel,
      where: { id: organizationModel.id },
    })
  }

  async findFirst(input: FindOrganizationProps): Promise
      <Organization | null> {
    const organization = await prisma.organizationModel.
        findFirst({
      where: {
        OR: [
          { id: input.id?.value }, { name: input.name?.
              value }
        ],
      },
    })

    if (!organization) return null

    return OrganizationMapper.toDomain(organization)
  }

  async exists(input: FindOrganizationProps): Promise<
      boolean> {
    const exists = await prisma.organizationModel.
        findFirst({
      where: {
        OR: [{ id: input.id?.value }, { name: input.name
            ?.value }],
      },
    })

    if (exists) return true

    return false
  }
}
```

Fonte: Do autor (2023)

Figura 4.10 – Interface do *AppleProvider*

```
export type LoadUserInput = {
  appleToken: AppleToken
}

export type LoadUserOutput = {
  appleId: AppleId
  email: Email
}

export interface AppleProviderInterface {
  loadUser(input: LoadUserInput): Promise<LoadUserOutput
      | null>
}
```

Fonte: Do autor (2023)

Assim como as demais, essa atividade incluiu o desenvolvimento de testes de unidade e de integração dos componentes desenvolvidos, que foram omitidos aqui apenas por limitações de espaço, já que seu código abrange uma grande quantidade de cenários e por isso é bastante extenso. Os testes desenvolvidos incluíram testes de unidade para a entidade, *mapper* e repositório *in-memory*, além de testes de integração para o repositório do *Prisma*.

Por fim, discutiremos uma última atividade que ao contrário das demais, interage diretamente com dependências externas.

4.4 Provider de autenticação da Apple

Essa funcionalidade consiste em um *provider* de autenticação social para contas *Apple*, que torna possível a autenticação do passageiro ou motorista na aplicação móvel usando *e-mails* providos pela *Apple*. Os casos de uso dos respectivos módulos utilizarão o *provider*, mais exatamente sua interface, para obter os dados do usuário no processo de cadastro.

A interface do *AppleProvider* é ilustrada na Figura 4.10. Ela define o formato dos dados de entrada e saída, além do método *loadUser* para obter as informações do usuário. Esse método recebe um *token*, e a partir dele obtém um identificador único do usuário e seu *e-mail*.

Na Figura 4.11 temos a implementação concreta da interface. O método *loadUser* recebe um *token* no formato *JSON Web Token* fornecido pela própria *Apple* e o decodifica. Dos dados obtidos, ele busca do cabeçalho um identificador de chave, e o utiliza para obter a chave pública da *Apple*. Finalmente, ele verifica a assinatura do *token* a partir da chave pública, garantindo a validade do *token* e dos dados contidos nele. A partir daí, são feitas verificações adicionais da validade dos dados, *value objetcs* são instanciados e retornados.

Essa atividade incluiu também o desenvolvimento de testes de unidade para o *provider*, omitidos aqui por limitações de espaço.

Figura 4.11 – *AppleProvider*

```
class AppleProvider implements AppleProviderInterface {
  private readonly baseUrl = 'https://appleid.apple.com'
  private readonly clientId = appConfig.apple.clientId

  constructor(
    private readonly httpClientProvider:
        HttpClientProviderInterface,
    private readonly jwtProvider: JwtProviderInterface
  ) {}

  async loadUser(input: LoadUserInput): Promise<
      LoadUserOutput | null> {
    const decodedAuthToken = this.jwtProvider.decode(
        input.appleToken.value)

    const { header } = decodedAuthToken
    const appleKeyIdentifier = header.kid

    const publicKey = await this.getPublicKey(
        appleKeyIdentifier)

    const userDataOrError = this.jwtProvider.verify(
        input.appleToken.value, publicKey)
    if (userDataOrError.isLeft()) return null

    const { sub, email, iss, aud } = userDataOrError.
        value

    if (!email) return null
    if (iss !== this.baseUrl) return null
    if (aud !== this.clientId) return null

    const appleId = AppleId.create(sub).value as AppleId
    const emailObject = Email.create(email).value as
        Email

    return { appleId, email: emailObject }
  }

  private async getPublicKey(appleKeyIdentifier: string)
      {
    const client = jwksClient({
      jwksUri: `${this.baseUrl}/auth/keys`,
      timeout: 30000,
    })

    const key = await client.getSigningKey(
        appleKeyIdentifier)

    return key.getPublicKey()
  }

export { AppleProvider }
```

Fonte: Do autor (2023)

4.5 Discussão

O estágio ofereceu uma valiosa experiência profissional, proporcionando não só a prática no desenvolvimento de software mas também uma visão mais ampla da qualidade e arquitetura de sistemas web.

O estágio abrangeu o desenvolvimento inicial de alguns módulos do sistema, incluindo o desenvolvimento do módulo *Organizations*, que permite que motoristas compartilhem véiculos e rotas, e por isso tem um importante papel no sistema. Atualmente o sistema se encontra mais próximo de sua primeira versão pública.

As atividades sempre foram atribuídas nas *sprints* de forma a ampliar o conhecimento de cada membro sobre o sistema, e podemos afirmar que essa estratégia foi oportuna e bem sucedida.

O ambiente de trabalho amigável e informal também foi importante, favorecendo o aprendizado e desenvolvimento profissional.

5 CONCLUSÃO

O estágio realizado na Buzu Brasil proporcionou o aprendizado e aplicação de boas práticas no desenvolvimento de software, com destaque para os princípios da Arquitetura Limpa e *design patterns*, discutidos na Seção 2.

O desenvolvimento de novas *features* (Seções 4.1, 4.3 e 4.4) favoreceu o entendimento do sistema, e o uso de padrões de projeto, como *providers* e *factories*, sempre observando as diretrizes da Arquitetura Limpa, em especial a regra de dependência e a separação de interesses. Já as refatorações realizadas (Seção 4.2) permitiram a aplicação de padrões de projeto como o *strategy*.

É possível afirmar que o estágio realizado na Buzu Brasil ofereceu uma oportunidade ímpar de aprendizado de boas práticas de desenvolvimento, desde os conceitos da Arquitetura Limpa e padrões de projeto, até o uso de ferramentas como testes automatizados e *pipelines* de integração e desenvolvimento contínuos.

Além disso, favoreceu uma perspectiva da engenharia de software com foco na qualidade e na arquitetura do sistema, onde tecnologias e *frameworks* são detalhes. Isso torna a experiência ainda mais valiosa comparada às experiências profissionais anteriores.

O entendimento e prática desses conceitos não seria possível sem os fundamentos obtidos no decorrer no curso de Ciência da Computação, em especial de disciplinas como Engenharia de Software e Práticas de Programação Orientada a Objetos.

REFERÊNCIAS

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. 4. ed. Boston: Addison-Wesley, 2021.

ETHEREDGE, J. **The Provider Model Pattern, Really?** 2009. Disponível em: <https://www.simplethread.com/the-provider-model-pattern-really/>.

EVANS, E. **Domain-Driven Design**: Tackling complexity in the heart of software. Boston: Addison-Wesley, 2003.

FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures**. Tese (Doutorado) — University of California, Irvine, 2000. Disponível em: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.

FOWLER, M. **Refactoring**: Improving the design of existing code. Boston: Addison-Wesley, 2018.

GAMMA, E. et al. **Design patterns**: Elements of reusable object-oriented software. Boston: Addison-Wesley, 1995.

HOWARD, R. **Provider Model Design Pattern and Specification**. 2006. Disponível em: <https://learn.microsoft.com/en-us/previous-versions/dotnet/articles/ms972319>.

LEMOS, O. **Arquitetura Limpa na prática**. Belo Horizonte: Hotmart, 2022.

MARTIN, R. **Clean Architecture**. Hoboken: Prentice Hall Press, 2017.

MARTIN, R. C. **The Clean Architecture**. 2012. Disponível em: <https://https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture>.

RED HAT. **Qual a diferença entre REST e SOAP?** 2009. Disponível em: <https://www.redhat.com/pt-br/topics/integration/whats-the-difference-between-soap-rest>.

RICHARDSON, L.; AMUNDSEN, M.; RUBY, S. **RESTful Web APIs**. Sebastopol: O'Reilly Media, 2013.

SCHWABER, J. S. K. **The Scrum Guide**. 2010. Disponível em: <https://scrumguides.org>.

TAYLOR, R. N.; MEDVIDOVIC, N.; DASHOFY, E. M. **Software Architecture**: Foundations, theory, and practice. Boston: Wiley, 2009.

VERNON, V. **Implementing Domain-Driven Design**. Boston: Addison-Wesley, 2013.