UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
INF01120 – TÉCNICAS DE CONSTRUÇÃO DE PROGRAMAS

Trabalho Prático - Etapa 2 - Grupo 5

Carolina Magagnin Wajner (00134101)
Gabriel Souza Lima (00584520)
Luísa Righi Bolzan (00578954)
Nicolas Chin Lee (00579322)
Rodrigo Salvadori Feldens (00578803)

Porto Alegre, Rio Grande do Sul

1. Introdução

Este relatório tem como objetivo apresentar a implementação e o teste da Etapa 2 do trabalho prático da disciplina INF01120. O foco é descrever o software desenvolvido, confrontando-o com o projeto inicialmente concebido na etapa anterior, detalhando as adaptações realizadas e as justificativas para tais decisões. Serão abordadas as classes implementadas, seus relacionamentos, os testes unitários aplicados e a interface do usuário desenvolvida.

2. Visão Geral do Projeto

O objetivo geral do projeto é desenvolver um gerenciador de Pokémons com recompensas no estilo clicker, denominado ***PokeClicker***. O aplicativo é voltado para um treinador (o usuário), que interage com o sistema por meio de "clicks" para adquirir recursos financeiros para a compra de itens e pokémons. A ideia é oferecer uma experiência de gerenciamento simplificada, onde a progressão e a aquisição de novos elementos são motivadas pela interação contínua do usuário.

3. Mudanças em Relação à Etapa Anterior

A estrutura do projeto original foi planejada com base nos princípios da Programação Orientada a Objetos (POO), buscando modularidade e extensibilidade. O design inicial se manteve consistente com uma classe abstrata **Pokémon**, da qual classes específicas para tipos de Pokémon (Fogo, Água e Grama) herdam comportamentos básicos. A classe **User** centraliza as ações do jogador, como clicar, navegar entre telas e gerenciar a coleção de pokémons e itens. As opções de tela, como **Shop** (Loja), **PC** (para visualização e manipulação de Pokémons) e **Home**, onde existem os **registros** de habilidades, itens e pokémons, além do Clicker, também foram modeladas como classes separadas.

3.1. Mudanças no Conceito Geral e Escopo

- **Mecânica Principal:** O foco do sistema mudou de uma ideia mais tradicional de Pokémons/jogo, que incluía um esboço de batalha, para foco exclusivo no estilo **clicker**.

Isso significa que a progressão e as interações do usuário são mais orientadas por ações de clique e aquisição de recursos por meio delas,

3.2. Modificações e Adaptações de Classes Existentes

As classes **User, Clicker, Item, Pokemon, Shop** e **PC** foram mantidas em grande parte, conforme o projeto original, mas tivemos a remoção da batalha. No entanto, algumas modificações significativas foram introduzidas para otimizar a funcionalidade e a extensibilidade, especialmente na interação entre as classes.

- Interfaces **Purchasable** e **Activable:**

    - **Definição Original:** Não havia interfaces explícitas para indicar que itens ou Pokémons poderiam ser comprados ou ativados.
    - **Implementação:** Foram criadas as interfaces Purchasable e Activable.
        - **Purchasable**: Define métodos relacionados à compra de um item, como isAvailable(): boolean, setAvailable(boolean): void e getPrice(): double.
        - Activable: Define uma classe que é ativável.
    - **Justificativa:** A introdução dessas interfaces foi necessária para generalizar o comportamento de elementos que podem ser comprados (itens e Pokémons na loja) ou ativados/utilizados. Isso promove o polimorfismo e simplifica a lógica da Shop, permitindo que ela lide de forma uniforme com diferentes tipos de produtos.

- **Herança na Classe Item:**
    - **Definição Original:** Originalmente a classe Item era um item genérico, sem a especificação do tipo da aplicação.
    - **Implementação:** A classe Item agora é uma classe pai abstrata. Foram criadas classes filhas específicas: MoneyMultiplierItem (filho) e PokemonItem (filho).
    - **Justificativa:** Essa mudança foi necessária para diferenciar e encapsular os comportamentos específicos de diferentes tipos de itens. Um MoneyMultiplierItem afeta a geração de dinheiro, enquanto um PokemonItem

pode estar relacionado à captura ou melhoria de Pokémons. Essa herança permite reutilizar o comportamento base de Item enquanto adiciona funcionalidades especializadas, seguindo o princípio da segregação de interfaces e responsabilidade única.

3.3. Classes Adicionadas

- **Classes de "Managers":**
    - **Definição Original:** Não estavam presentes no projeto inicial.
    - **Implementação:** Foram adicionadas classes do tipo "managers" (gerenciadores) para as classes principais (User, Pokémon, Item).
    - **Justificativa:** A adição dessas classes (UserManager, PokemonManager, ItemManager, PCManager) centraliza a lógica de negócios e as operações relacionadas a essas entidades. Isso segue o princípio da separação de responsabilidades (SRP), tornando o código mais organizado, manutenível e escalável. Ao isolar as operações de manipulação de dados para cada tipo de entidade, evita-se a desorganização e concentração de lógica em outras classes, que deveriam focar em suas próprias responsabilidades.

3.4. Classes Removidas

- **Classe Batalha:**
    - **Definição Original:** Prevista no projeto inicial para ser a maneira de evoluir e ganhar dinheiro no gerenciador.
    - **Implementação:** Foi removida.
    - **Justificativa:** A funcionalidade de "Batalha" não foi implementada e não se encaixou no escopo simplificado e focado na mecânica clicker do trabalho atual. A

remoção visa manter o projeto alinhado com os requisitos priorizados para esta etapa, evitando complexidade desnecessária.

3.5. Alterações nos Relacionamentos entre Classes

- **Relação Ability com Pokemon:**
    - **Definição Original:** Ability era uma agregação, sugerindo que poderia existir independentemente e ser associada a um Pokémon.
    - **Implementação:** Ability agora é uma **dependência** de Pokémon.
    - **Justificativa:** A mudança de agregação para dependência significa que uma Ability não pode mais ser criada sozinha; ela é criada e diretamente associada a um Pokémon no momento da sua criação. Isso reflete uma relação mais forte de ciclo de vida e responsabilidade: a existência e a manipulação de uma Ability estão intrinsecamente ligadas ao seu pokémon associado, simplificando a gestão e garantindo que habilidades não existam sem um contexto de uso.

4. Detalhes da Implementação

Esta seção detalha as classes implementadas no projeto, incluindo seus atributos, métodos, parâmetros, retornos e modificadores de acesso.

LINK PARA O DIAGRAMA ATUALIZADO (todas as classes, métodos e relacionamentos): https://lucid.app/lucidchart/9da4de9f-e237-4ddf-bf41-82c79efb18f7/edit?page=0_0#

No link acima é possível compreender todas as classes do trabalho, bem como abstrações e interfaces. Além disso, estão claros os métodos, atributos e relacionamentos entre as classes, podendo dar zoom para observar melhor. Está em formato de link, dada a impossibilidade de visualização em uma imagem única por conta do seu tamanho.

4.1. Linguagem de Programação e Tecnologias Utilizadas

O projeto foi implementado utilizando a linguagem de programação **Java** , seguindo o paradigma de Programação Orientada a Objetos. Para o desenvolvimento da interface gráfica, foi empregado o **JavaFX**. O **SQLite** foi utilizado para o armazenamento de dados.

4.2. Classes Principais Implementadas

As principais classes implementadas são: **User, Clicker, Item, Pokemon, PC e Shop**, além das interfaces e classes de gerenciamento já mencionadas. Abaixo, são detalhados os atributos e métodos **não óbvios** de cada uma, focando em decisões de design relevantes.

4.2.1. Classe User

Representa o treinador do jogo e suas características principais.

- **Atributos:**
    - private double moneyMultiplier: Multiplicador de dinheiro que o usuário possui.
    - private double favoritePokemon: Pokémon favorito.
- **Métodos (Exemplos, além de getters e setters padrão):**
    - public void spendMoney(double): Gastar uma quantidade de dinheiro ao comprar.
    - public Pokemon getFavoritePokemon(): Retorna o pokemon favorito do usuário.

4.2.2. Classe Clicker

Gerencia a lógica de recompensas baseada em cliques.

- **Atributos:**
    - private int moneyPerClick: Valor base de dinheiro gerado por clique.
    - private int totalClicks: Contador de cliques realizados.
- **Métodos (Exemplos, além de getters e setters padrão):**
    - public void setMoneyPerClick(in moneyPerClick): Define a recompensa total de dinheiro para um clique, considerando o multiplicador.
    - public User registerClick(): Registra o click para o User atual.

4.2.6. Classe Pokemon

Classe abstrata que representa um Pokémon genérico.

- **Atributos:**
    - private LevelType level: Nível do Pokémon.
    - private List<Ability> abilities: Lista de habilidades do Pokémon.
- **Métodos (Exemplos, além de getters e setters padrão):**
    - public boolean updateLevel(): aumentar o level do pokémon.
- **Herança:** Serve como classe pai para tipos específicos de Pokémons (ex: FirePokemon, WaterPokemon, GrassPokemon).

4.2.7. Classe Shop

Gerencia a venda de itens e Pokémons.

- **Atributos:**
    - private final User user: User da shop
- **Métodos (Exemplos, além de getters e setters padrão):**
    - public Purchasable buyPokemonOrItem(Purchasable): comprar um pokémon ou item

4.2.10. Classe SceneSwitcher

- **Propósito:** Gerenciar a transição entre diferentes cenas (telas) na interface JavaFX.
- **Métodos:**
    - public static void switchToProfile(Action event, String username): Método de classe (estático) que carrega um arquivo FXML e define-o como a cena atual. Também seta o usuário como contexto para a próxima cena
    - **Justificativa para Métodos Estáticos:** Os métodos estáticos no SceneSwitcher são justificados porque a funcionalidade de troca de cenas é uma operação global e intrínseca à arquitetura do JavaFX, sendo acessada por diversas classes de controladores de tela. Utilizar métodos estáticos simplifica o acesso a essa

funcionalidade sem a necessidade de instanciar a classe SceneSwitcher em múltiplos locais.

5. Teste

Esta parte do trabalho descreve os testes unitários automatizados realizados no projeto. Os testes foram projetados e implementados durante o desenvolvimento do código , utilizando a ferramenta **JUnit**. O objetivo foi garantir a funcionalidade e a robustez das classes do domínio.

5.1. Detalhamento dos Testes Unitários e Casos de Teste

Os testes unitários abrangeram obrigatoriamente todas as classes do domínio do problema, como User, Clicker, Item (e suas subclasses), Pokemon (e suas subclasses), Shop, e as classes de "Managers". Classes relacionadas à interface (JavaFX) e ao armazenamento de dados (SQLite) não foram testadas unitariamente, seguindo as diretrizes do trabalho.

A seguir, são apresentados exemplos de casos de teste para as classes principais:

5.1.1. Testes para a Classe User

- **testAddMoney():**
    - **Cenário:** Adicionar dinheiro a um usuário com saldo inicial.
    - **Ações:** Criar um User com money = 0. Chamar addMoney(100).
    - **Resultado Esperado:** user.getMoney() deve retornar 100.
- **testSpendMoneySufficientFunds():**
    - **Cenário:** Gastar dinheiro com saldo suficiente.

- **Ações:** Criar um User com money = 500. Chamar spendMoney(200).
- **Resultado Esperado:** O método deve retornar true. user.getMoney() deve retornar 300.

- **testSpendMoneyInsufficientFunds():**
  - **Cenário:** Tentar gastar dinheiro com saldo insuficiente.
  - **Ações:** Criar um User com money = 100. Chamar spendMoney(200).
  - **Resultado Esperado:** O método deve retornar false. user.getMoney() deve permanecer 100.
- **testAddPokemon():**
  - **Cenário:** Adicionar um Pokémon à lista do usuário.
  - **Ações:** Criar um User. Criar uma instância de Pokemon. Chamar addPokemon(pokemon).
  - **Resultado Esperado:** user.getPokemons().size() deve ser 1. A lista deve conter o Pokémon adicionado.
- **testAddItem():**
  - **Cenário:** Adicionar um item ao inventário do usuário.
  - **Ações:** Criar um User. Criar uma instância de Item (ex: MoneyMultiplierItem). Chamar addItem(item).
  - **Resultado Esperado:** user.getItems().size() deve ser 1. O inventário deve conter o item adicionado.

5.1.2. Testes para a Classe Clicker

- **testCalculateClickRewardBase():**
  - **Cenário:** Calcular recompensa com multiplicador padrão.
  - **Ações:** Criar um Clicker com baseMoneyPerClick = 10 e moneyMultiplier = 1.0. Chamar calculateClickReward().
  - **Resultado Esperado:** Retornar 10.0.
- **testCalculateClickRewardWithMultiplier():**
  - **Cenário:** Calcular recompensa com multiplicador aplicado.
  - **Ações:** Criar um Clicker com baseMoneyPerClick = 10 e moneyMultiplier = 2.5. Chamar calculateClickReward().

    - **Resultado Esperado:** Retornar 25.0.
- **testApplyMoneyMultiplier():**
    - **Cenário:** Aplicar um novo multiplicador.
    - **Ações:** Criar um Clicker com moneyMultiplier = 1.0. Chamar applyMoneyMultiplier(0.5).
    - **Resultado Esperado:** clicker.getMoneyMultiplier() deve retornar 1.5. (Assumindo que o método soma o valor).
- **testIncrementClicks():**
    - **Cenário:** Incrementar o contador de cliques.
    - **Ações:** Criar um Clicker com clicksCount = 0. Chamar incrementClicks() duas vezes.
    - **Resultado Esperado:** clicker.getClicksCount() deve retornar 2.

5.1.3. Testes para as Classes Item e suas Subclasses (MoneyMultiplierItem, PokemonItem)

- **testMoneyMultiplierItemUse():**
    - **Cenário:** Usar um item que aumenta o multiplicador de dinheiro.
    - **Ações:** Criar um User e um Clicker associado. Criar um MoneyMultiplierItem com multiplierValue = 0.2. Chamar item.use(user).
    - **Resultado Esperado:** user.getClicker().getMoneyMultiplier() deve ter sido aumentado em 0.2 (ou conforme a lógica de aplicação do item).
- **testPokemonItemUse():**
    - **Cenário:** Usar um item que adiciona um Pokémon.
    - **Ações:** Criar um User. Criar um Pokemon de teste. Criar um PokemonItem com o Pokémon de teste. Chamar item.use(user).
    - **Resultado Esperado:** user.getPokemons().size() deve ser 1. A lista de Pokémons do usuário deve conter o Pokémon de teste.

5.1.4. Testes para a Classe Shop

- **testAddProduct():**
    - **Cenário:** Adicionar um item comprável à loja.

    - **Ações:** Criar uma Shop. Criar uma instância de Item (que implementa Purchasable). Chamar shop.addProduct(item).
    - **Resultado Esperado:** shop.getAvailableProducts().size() deve ser 1. A lista deve conter o item adicionado.
- **testPurchaseProductSuccessful():**
    - **Cenário:** Comprar um produto com dinheiro suficiente.
    - **Ações:** Criar um User com dinheiro. Criar um Shop e adicionar um Purchasable com preço. Chamar shop.purchaseProduct(user, product).
    - **Resultado Esperado:** O método deve retornar true. O dinheiro do usuário deve ser subtraído. O item deve ser adicionado ao inventário do usuário.
- **testPurchaseProductInsufficientFunds():**
    - **Cenário:** Tentar comprar um produto sem dinheiro suficiente.
    - **Ações:** Criar um User com pouco dinheiro. Criar um Shop e adicionar um Purchasable com preço alto. Chamar shop.purchaseProduct(user, product).
    - **Resultado Esperado:** O método deve retornar false. O dinheiro do usuário e o inventário devem permanecer inalterados.

5.2. Experiência com Testes Unitários

A experiência do grupo em utilizar testes unitários durante o desenvolvimento (ou antes, seguindo princípios de TDD) foi extremamente valiosa. A abordagem de testes automatizados com JUnit permitiu identificar e corrigir bugs precocemente no ciclo de desenvolvimento, antes que se propagasse para outras partes do sistema. Isso resultou em um código mais robusto e com menos falhas, facilitando a refatoração e a adição de novas funcionalidades com confiança. A obrigatoriedade de testar todas as classes do domínio garantiu uma cobertura abrangente das funcionalidades essenciais, contribuindo para a estabilidade geral da aplicação. A prática de TDD, mesmo que não estritamente seguida em todos os momentos, influenciou positivamente a forma como as classes foram projetadas, buscando maior modularidade e testabilidade desde o início.

6. Executável (Aplicação, Interface)

Esta seção apresenta detalhes sobre a aplicação executável, as funcionalidades desenvolvidas e sua interface, acompanhadas de imagens do software em execução. Embora a prioridade do trabalho não seja a interface, foi desenvolvida uma interface gráfica simples para permitir a interação do usuário.

6.1. Manual de Uso Simplificado

Para executar o aplicativo PokéClicker:

1. **Pré-requisitos:** Certifique-se de ter o Java Runtime Environment (JRE) ou Java Development Kit (JDK) versão 11 ou superior instalado em sua máquina.
2. **Download:** Obtenha o arquivo PokéClicker.jar e os recursos necessários (se houver, como imagens ou arquivos de banco de dados SQLite) dentro do arquivo executavel.zip fornecido na entrega.
3. **Execução:** Para executar a aplicação no VS Code, navegue até a visualização Executar e Depurar (o ícone geralmente se parece com um bug com um botão de "play"). Certifique-se de que a aba "Launch" esteja selecionada e clique no botão verde de "play" para iniciar a aplicação.

Como Jogar

- Para começar, insira o nome de usuário desejado.

- Após fazer login, você será direcionado para sua Página do Jogador. Nela, você pode visualizar seus Pokémons favoritos e seu saldo atual de dinheiro. Você também pode fazer logout a partir desta tela.

- Navegue até a Página Inicial (Home Page) para encontrar quatro atividades principais: Cadastro de Itens, Cadastro de Pokémons, Clicker e Cadastro de Habilidades.

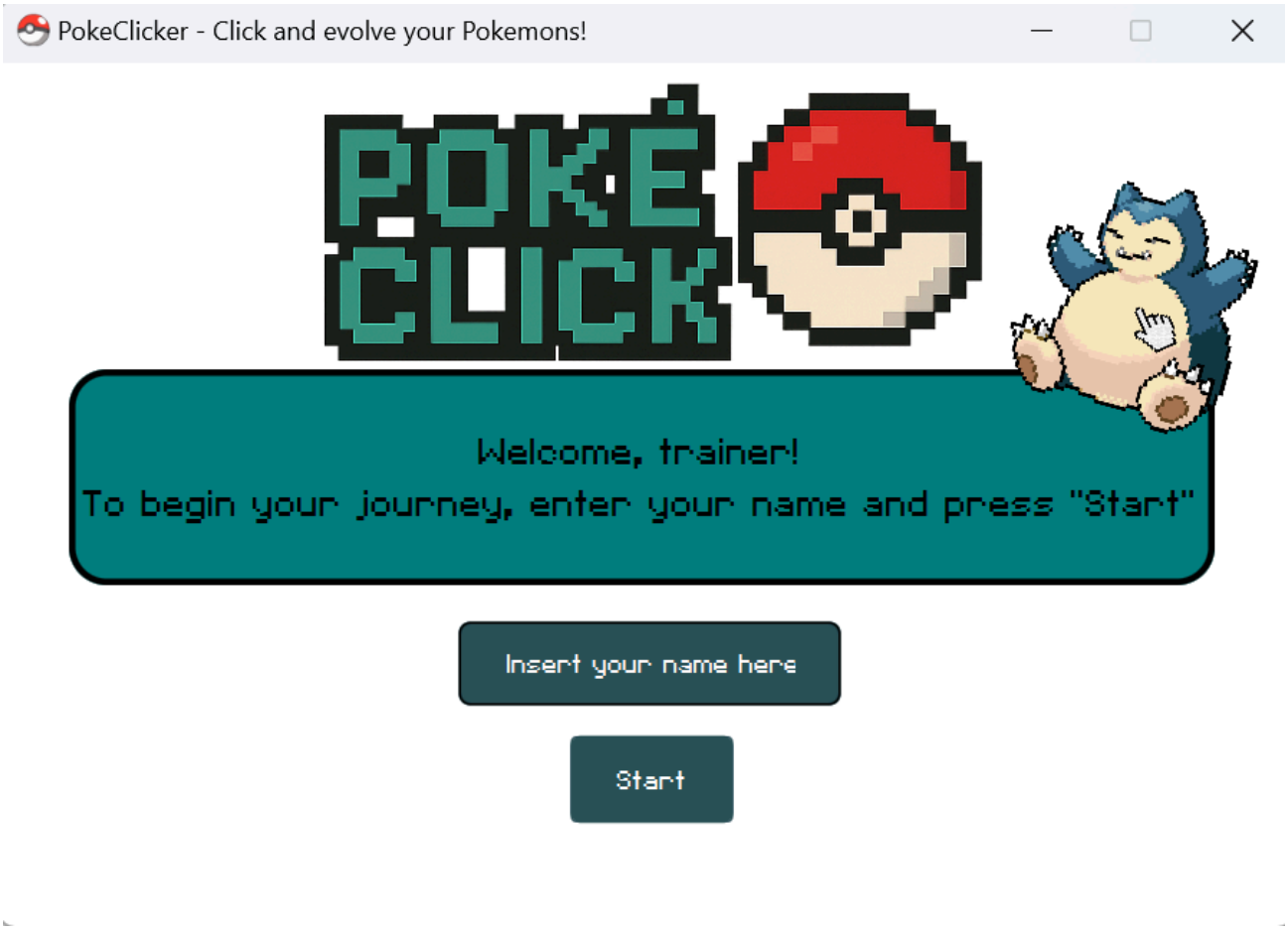
Para cadastrar um novo item, preencha o formulário com seus detalhes e selecione um tipo de item, conforme mostrado nos exemplos abaixo.

O item recém-cadastrado aparecerá agora no seu PC (Inventário/Caixa de Pokémons).

- Da mesma forma, para cadastrar um novo Pokémon, você deve preencher o formulário com seus detalhes, selecionar seu tipo e fazer o upload de uma imagem para ele, conforme o exemplo. Uma vez cadastrado, o Pokémon será adicionado à sua coleção, que também pode ser visualizada no PC.

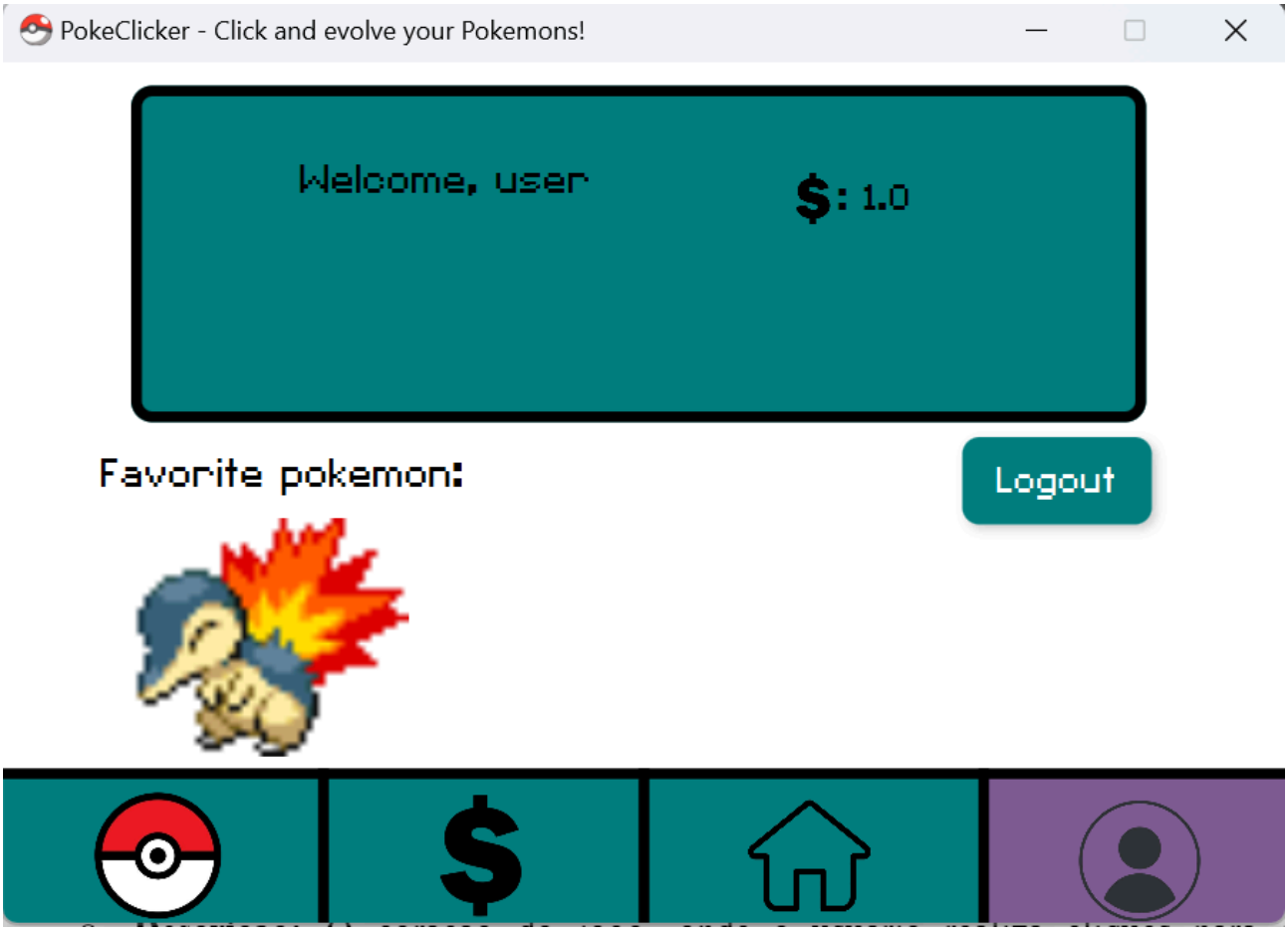
6.2. Funcionalidades e Interfaces

- **Tela Inicial / Login:**



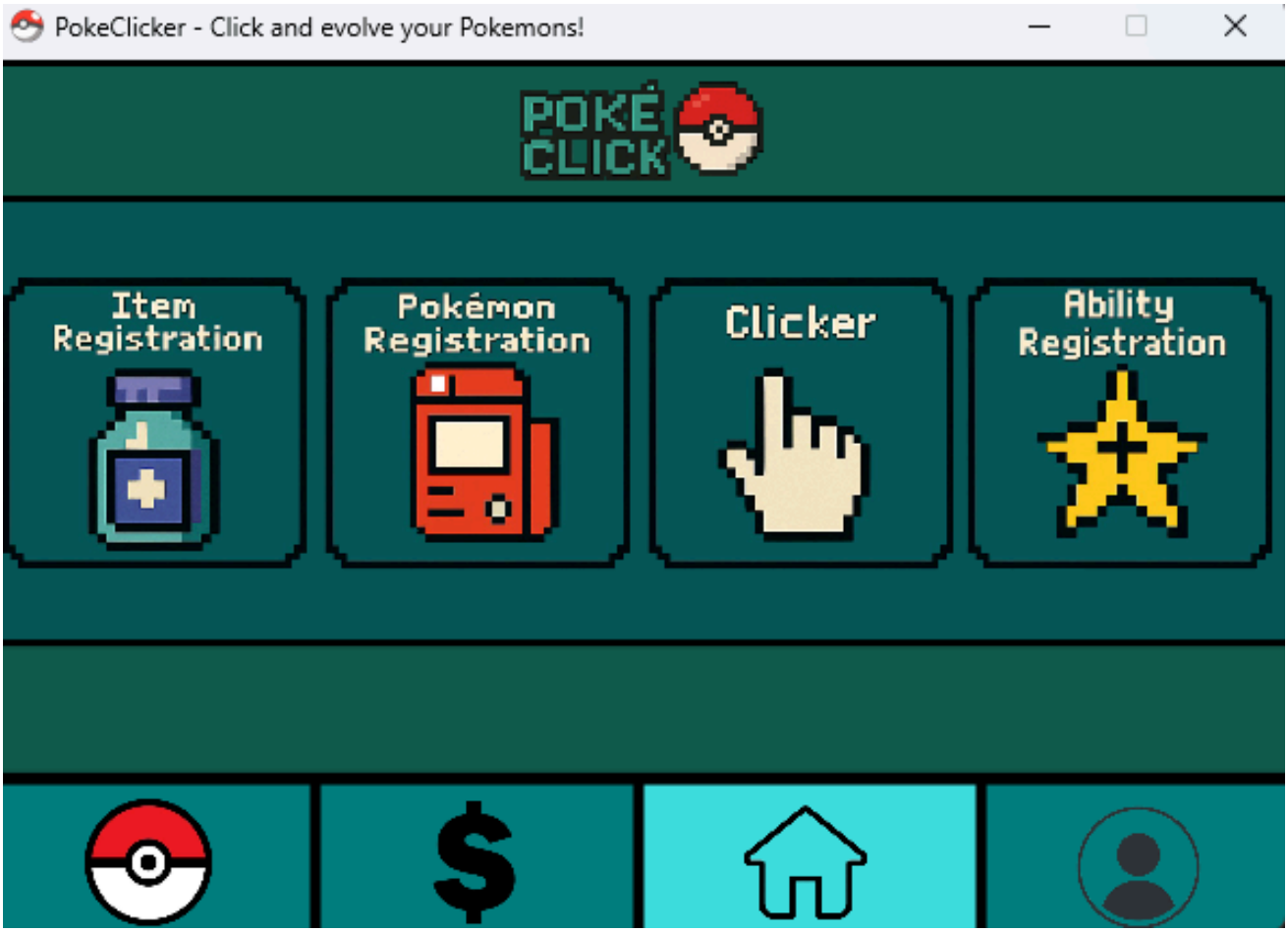
- **Descrição:** Tela de boas-vindas que permite ao usuário iniciar um novo jogo (novo nome) ou carregar um jogo existente (nome já utilizado).
- **Interações:** Botão "Start" cria/log um User a partir do input.

- **Profile:**



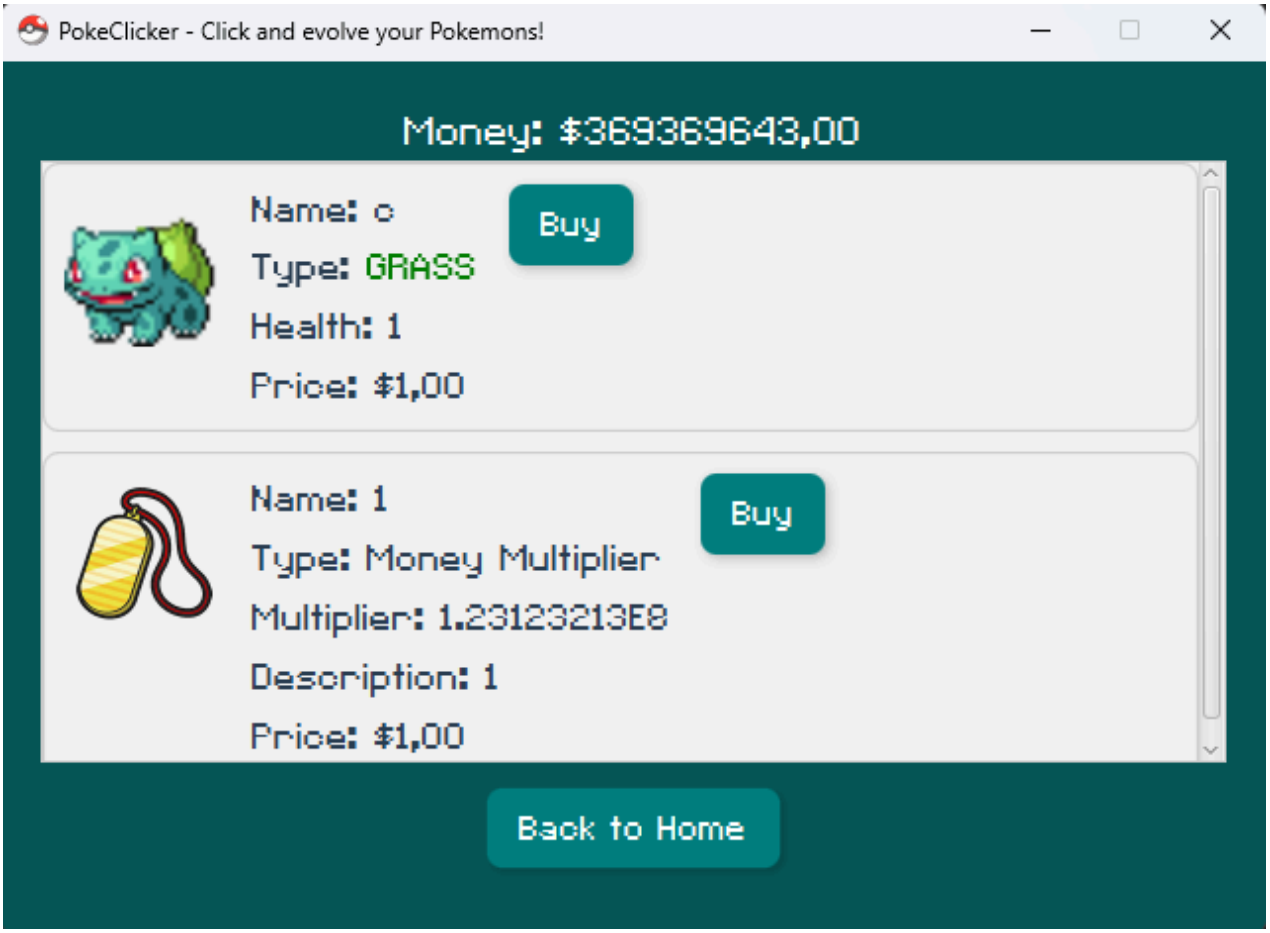
- **Descrição:** Dados do user.
- **Interações:**
  - **Botão "Logout"**: Ao ser clicado, user volta para a cena inicial.
  - **Exibição de Dinheiro:** Um label ou texto mostrando o dinheiro.
  - **Botão "Shop"**: Navega para a tela da Shop.
  - **Botão "Home"**: Navega para a tela da Home.
  - **Botão "PC"**: Navega para a tela de gerenciamento de Pokémons e itens (PC).

- **Home**



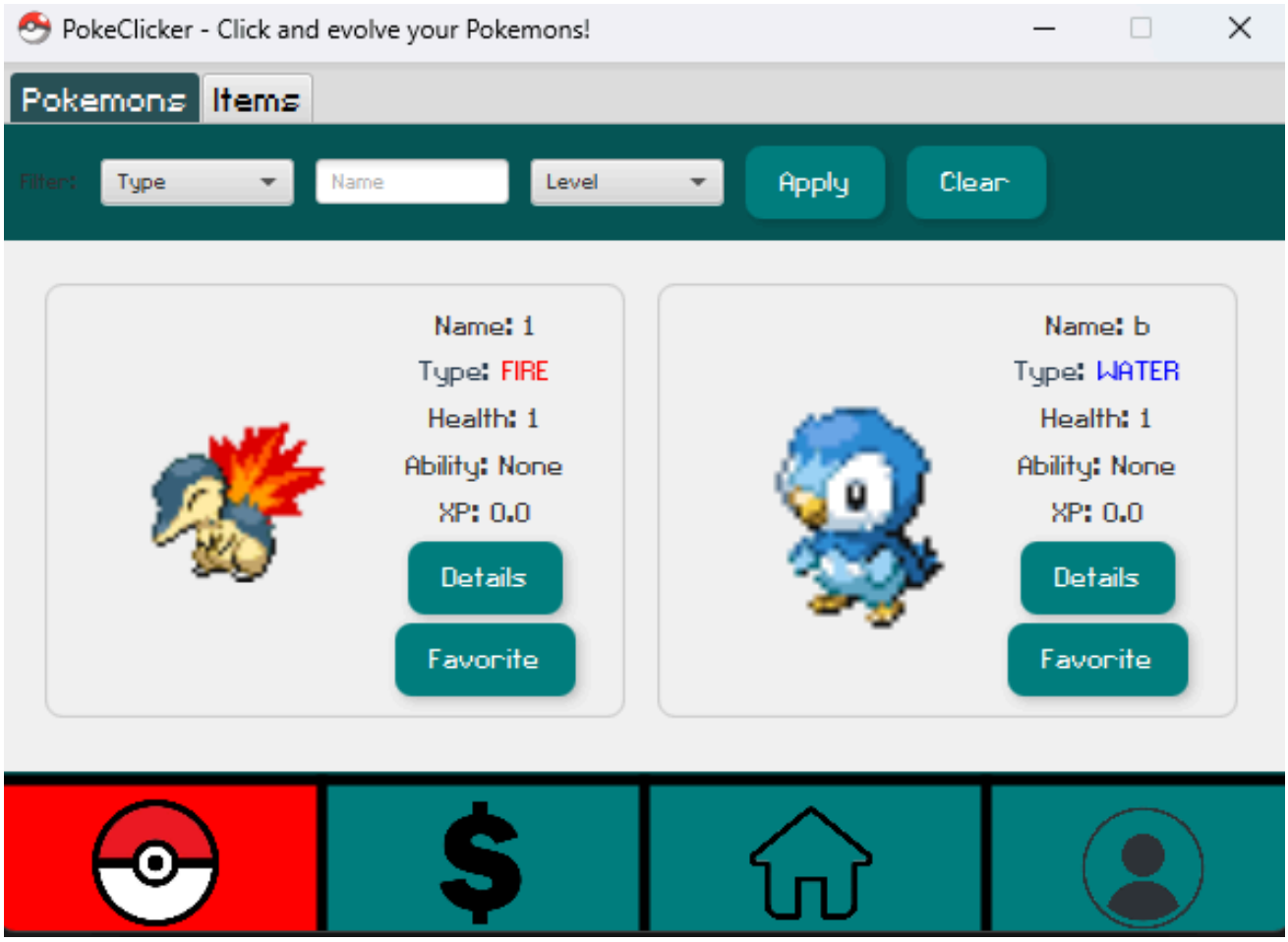
- **Lista de ações de registro:** Criar itens, pokémons e habilidades. Há também a opção do jogo de clicker.

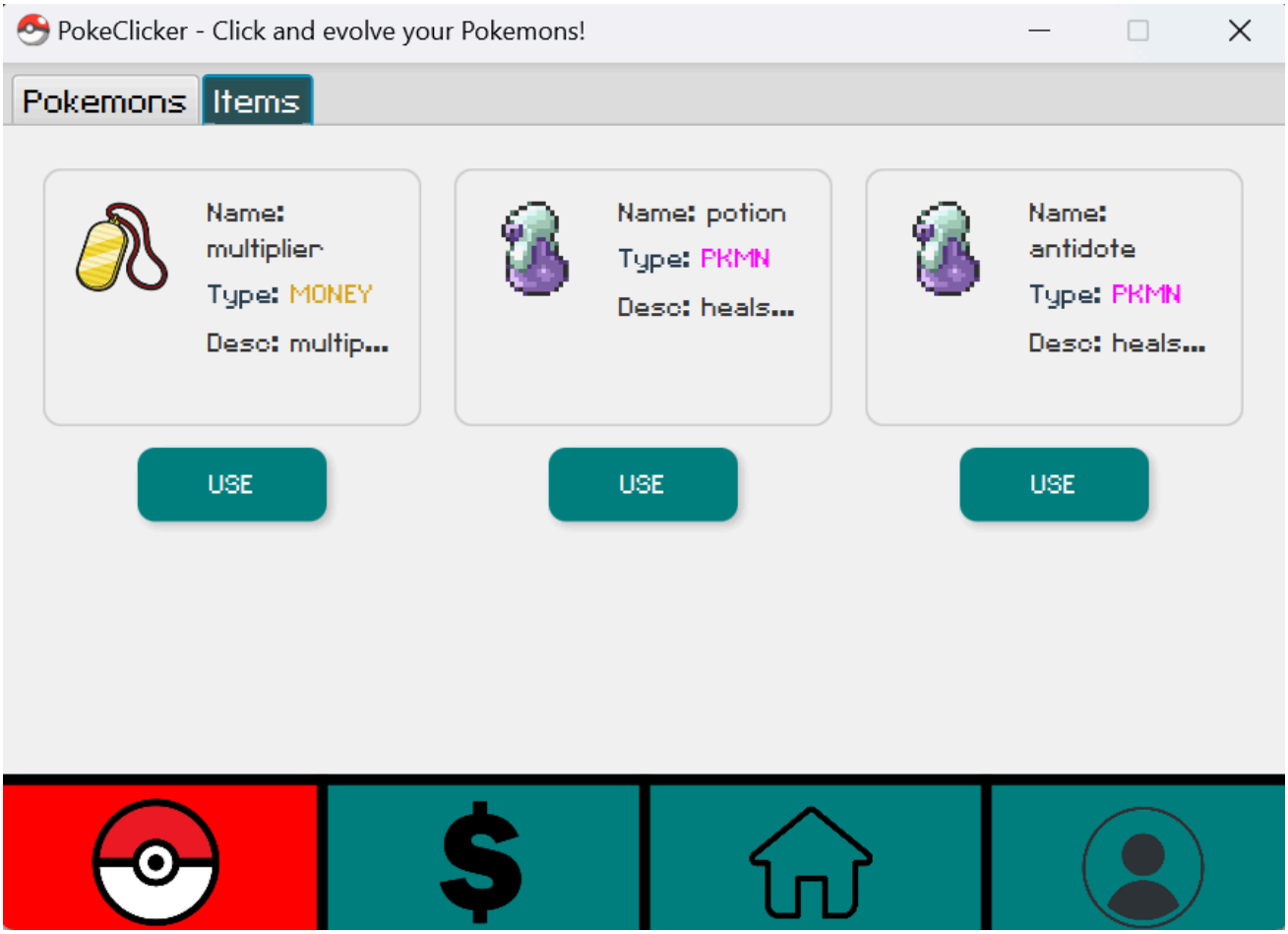
- **Shop:**



- **Descrição:** Permite ao usuário visualizar e comprar itens e Pokémons disponíveis.
- **Interações:**
  - **Botões "Comprar"**: Ao clicar no botão de um produto, a lógica da Shop.purchaseProduct() é acionada, deduzindo o dinheiro do usuário e adicionando o item/Pokémon ao seu inventário.
  - **Botão "Voltar"**: Retorna à tela principal.

- **PC:**


PokeClicker - Click and evolve your Pokemons!
Pokemons
Items
Filter:
Type
Name
Level
Apply
Clear
Name: 1
Type: FIRE
Health: 1
Ability: None
XP: 0.0
Details
Favorite
Name: b
Type: WATER
Health: 1
Ability: None
XP: 0.0
Details
Favorite



- **Descrição:** Exibe a coleção de Pokémons e Itens do usuário. Permite visualizar detalhes e, futuramente, interagir com ele.
- **Interações:**
  - **Lista de Pokémons e itens**
  - **Detalhes do Pokémon:** Ao selecionar um Pokémon, pode exibir seu nome, nível e habilidades.