

Projeto Interdisciplinar Para Sistemas De Informação 1

MUTARE: O Impulso Para Mudar

Pedro A. A. da Cunha¹, Maria L. L. Cordeiro¹

¹Departamento de Informática e Estatística (DEINFO)

Universidade Federal Rural de Pernambuco Sede (UFRPE Sede)

Estrada dos Macacos, 95, Dois Irmaos, Recife - PE, 52171-235, Brazil

pedroailton630@gmail.com, lauracordeiro259@gmail.com

Abstract. *Mutare is a safe tool for conscious habit management that seeks to bring quality of life, health and productivity in the current era of incessant stimuli arising from the amenities of new technologies, which discourages the practice of good habits. From the powerful psychology of habit (accurately treated by Charles Duhigg in his book "The Power of Habit", inspiration for our project), we created a digital system capable of providing assistance in the development of new habits (which the system can also suggest - such as sustainable and citizen habits), correction of bad habits and monitoring of habits developed by users, along with a system of rewards, performance measurement and mascot.*

Resumo. *O Mutare é uma ferramenta segura de gerenciamento consciente de hábitos que busca trazer qualidade de vida, saúde e produtividade na atual era de estímulos incessantes advindos das comodidades das novas tecnologias, que desincentiva a prática de bons hábitos. A partir da poderosa psicologia do hábito (assertivamente tratada por Charles Duhigg em seu livro "O Poder do Hábito", inspiração para o nosso projeto), criamos um sistema digital capaz de fornecer assistência ao desenvolvimento de hábitos novos (que o sistema também poderá sugerir - como hábitos sustentáveis e cidadãos), correção de maus hábitos e acompanhamento dos hábitos desenvolvidos pelos usuários, junto de um sistema de recompensas, medição de desempenho e mascote.*

1. Introdução

1.1. Contexto

A proposta da disciplina de Projetos Interdisciplinares para Sistemas de Informação 1 (PISI 1) do curso de Bacharelado em Sistemas de Informação na UFRPE, com docência do professor Cleiton Vanut Cordeiro de Magalhães, foi a solução de um problema da realidade de escolha dos desenvolvedores-autores por meio de um sistema digital - o docente propôs que fossem preferencialmente problemas de sustentabilidade, ética psicologia ou sociedade.

A equipe optou por abordar um campo situado entre as dimensões social e psicológica, a partir da proposta do desenvolvedor Pedro Ailton, inspirada na leitura do livro O Poder do Hábito, de Charles Duhigg, ex-repórter do The New York Times [Duhigg 2012]. Na obra, o autor destaca a relevância do gerenciamento de hábitos como

um fator essencial para a construção de uma vida mais saudável e significativa. Embora desafiador, esse processo é apresentado como viável e aplicável tanto no cotidiano individual quanto nos contextos sociais e organizacionais, por meio da compreensão e reestruturação de padrões comportamentais.

Ao longo do livro, Duhigg recorre a diversos estudos de caso para demonstrar a influência dos hábitos sobre os pilares fundamentais da sociedade contemporânea (a vida pessoal, os negócios e a sociedade). Um exemplo emblemático é o de Bill Wilson, fundador dos Alcoólicos Anônimos, cuja trajetória ilustra como a mudança de hábitos pode desempenhar um papel central na recuperação de dependentes. Mesmo sem embasamento científico formal em sua origem, a organização se consolidou como um dos programas de reabilitação mais eficazes do mundo, sustentando-se principalmente na força dos hábitos sociais e na substituição de rotinas prejudiciais por práticas mais saudáveis.

1.2. Justificativa

Índices crescentes de problemas relacionados à falta de gestão da própria rotina e hábitos no dia a dia da sociedade atual; como, por exemplo, dados da Associação Brasileiro do Sono (ABS) dizem que 73 milhões de brasileiros em 2023 sofriam de insônia, e hábito de interagir com dispositivos eletrônicos emissores de luz antes dormir pode ser agravante [G1 - Globo 2023].

Essa rotina vem ficando mais frequente, como informa o artigo [Anjos 2020], de Karlyne Maciel Gadêlha dos Anjos (2020), pesquisa realizada com adolescentes de áreas urbanas e suburbanas, aponta que o uso intenso de dispositivos eletrônicos, sobretudo no período noturno, compromete tanto a qualidade quanto a duração do sono. Essa alteração nos padrões de sono, por sua vez, está relacionada a prejuízos significativos na atenção e no desempenho escolar.

Além de desconhecimento da simplicidade de aplicar o poder do hábito para controlá-los e viver uma vida melhor, em saúde e propósito, tendo em vista na internet coaches e influencers que incentivam estilos de vida que estão fora da realidade da maioria da população (por razões financeiras, de tempo etc.).

1.3. Solução

A solução foi o sistema Mutare, um sistema digital capaz de:

1. Fornecer assistência ao desenvolvimento de hábitos novos (que o sistema também poderá sugerir - como hábitos sustentáveis e cidadãos), com possibilidade de acompanhamento diário, perfil (por meio de cadastro) e recompensas no sistema;
2. Acompanhamento dos hábitos desenvolvidos pelos usuários, junto de um sistema de recompensas, medição de desempenho e mascote;
3. Disseminação de conhecimento para a consciente utilização do programa, e gerenciamento de hábitos até mesmo fora dele.

Por meio da conscientização, demonstração e acompanhamento, o projeto é capaz de mudar a vida das pessoas, empreendimentos e, principalmente, a sociedade.

2. Metodologia

2.1. Etapas

2.1.1. Problemática

A etapa inicial do projeto consistiu na identificação de uma problemática real, relevante e impactante para a sociedade, servindo como ponto de partida para o levantamento de requisitos do sistema. Após discussões em grupo, análise de dados e observação do cotidiano, identificou-se que muitas pessoas enfrentam dificuldades em manter uma rotina equilibrada, produtiva e saudável. Problemas como falta de organização, hábitos nocivos (como uso excessivo de telas), insônia, procrastinação e sensação de vazio foram identificados como recorrentes, especialmente em um contexto urbano e superconectado.

Além disso, foi observado que, embora existam diversas ferramentas tecnológicas voltadas à produtividade e bem-estar, muitas delas são complexas, não personalizadas ou deixam de considerar aspectos motivacionais e comportamentais dos usuários.

Essa lacuna entre a necessidade de mudança de hábitos e as ferramentas realmente eficazes para isso foi o que fundamentou a escolha da problemática a ser abordada. O desafio seria, então, desenvolver um sistema que auxilie os usuários na construção de hábitos positivos e sustentáveis, oferecendo suporte, acompanhamento e motivação de forma acessível, lúdica e personalizada.

A partir dessa definição inicial, iniciou-se o levantamento de requisitos funcionais e não funcionais do sistema *Mutare*, com base na realidade do público-alvo e nas propostas de solução mais adequadas ao contexto identificado.

2.1.2. Design do Projeto

Nesta etapa, foi realizado o mapeamento dos fluxos de uso do sistema, com a criação de fluxogramas que contemplassem tanto os caminhos principais quanto os fluxos alternativos e de erro. Essa fase foi essencial para a compreensão integral do comportamento esperado do sistema diante de diferentes situações de interação do usuário.

Para o desenvolvimento dos fluxogramas, utilizou-se a ferramenta *draw.io*, uma plataforma gratuita que permite a criação de diagramas e exportação em formatos como o usado PDF. Um dos principais desafios enfrentados durante essa fase foi a modelagem dos fluxos de erro, ou seja, os caminhos que representam comportamentos inesperados ou incorretos por parte do usuário. Dentre os casos mais recorrentes, destacou-se o tratamento de erros de digitação durante a navegação pelo menu, especialmente enquanto a interface do sistema *Mutare* ainda estava restrita ao terminal. Segue abaixo um recorte do fluxograma desse tratamento de fluxo de erro aplicado ao fluxograma da funcionalidade que direciona o programa para as outras, o *Menu Principal*:

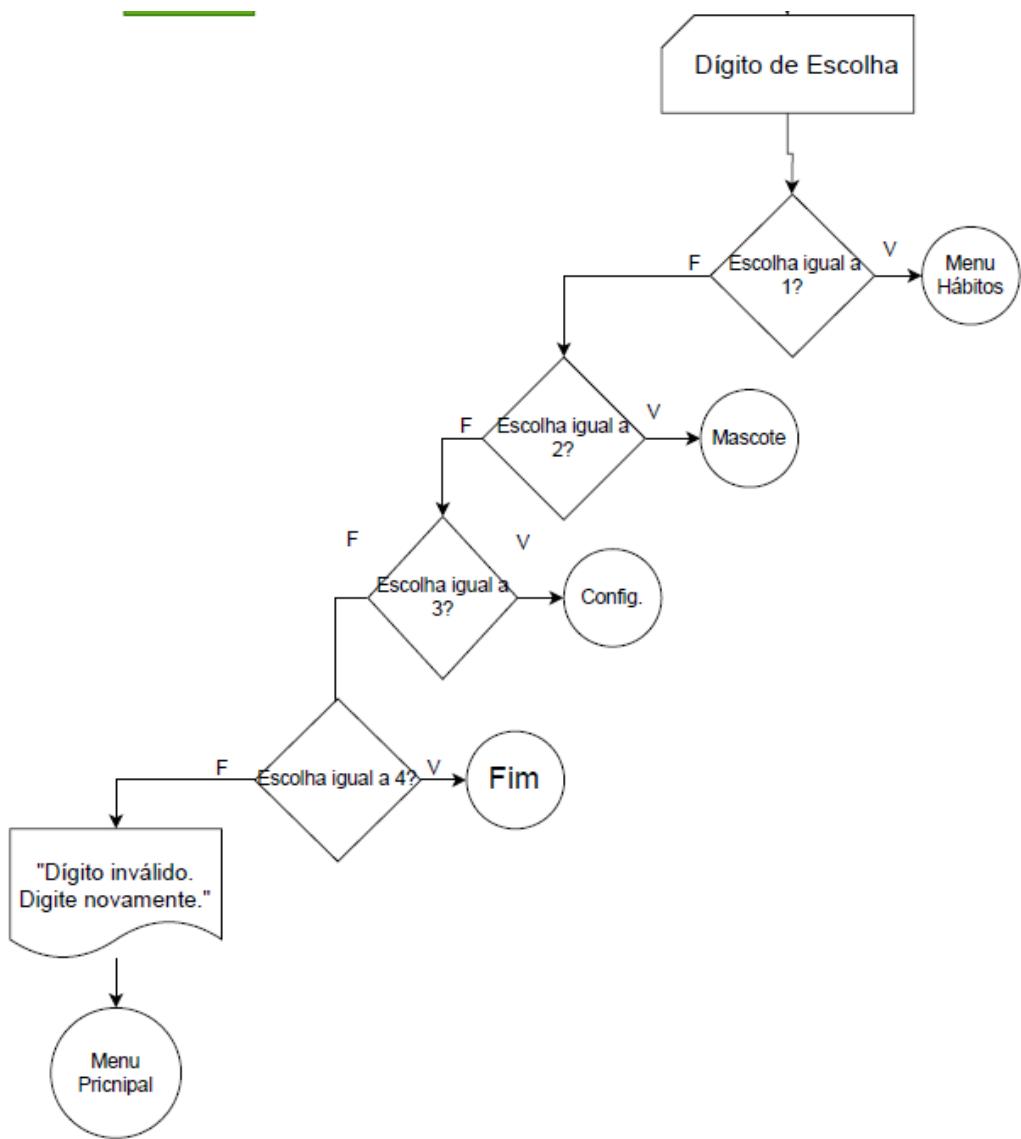


Figure 1. Arquivo em PDF do Fluxograma da Funcionalidade Menu Principal

Como é possível observar, há uma cascata de estrutura de decisão (a imagem do losango), onde, cada condição verdadeira leva para uma outra etapa do programa, enquanto uma condição falsa desencadeia em outra decisão. Quando nenhuma condição é verdadeira, só há a possibilidade da entrada ter sido inválida, então há esse aviso do programa para o usuário, e o retorno para o início do Menu Principal.

2.1.3. Implementação do Projeto

Com os fluxos definidos, iniciou-se a etapa de implementação, utilizando a linguagem de programação Python. Para a 1^a Release (lançamento) do sistema, o paradigma da programação funcional foi utilizado, dada a sua simplicidade de entendimento e o desenvolvimento teórico e prático dos desenvolvedores na tecnologia, já que envolve apenas a definição de funções e utilização delas. A partir da 2^a release, o projeto foi modelado para

o paradigma do Programa com Orientação ao Objetos (POO) e foi modularizado (código separado em diferentes arquivos).

Foram empregadas bibliotecas como `customtkinter`, para a construção da interface gráfica, e outras dependências auxiliares conforme a necessidade do projeto.

Além disso, a ferramenta Git foi utilizada para versionamento e manutenção do repositório, assegurando o controle das alterações e facilitando a colaboração entre os membros da equipe.

2.1.4. Testes

Na etapa de testes, foi realizada a validação dos fluxos implementados com base nos requisitos previamente definidos. Essa validação consistiu em conduzir o sistema por diferentes caminhos, de modo a simular os diversos cenários previstos durante a etapa de design, incluindo tanto os fluxos principais quanto os alternativos e os de erro. O objetivo foi garantir que o comportamento do sistema estivesse plenamente alinhado às necessidades identificadas no planejamento, assegurando que cada funcionalidade respondesse de forma adequada às interações do usuário.

Durante o processo, os testes não apenas confirmaram o funcionamento esperado do sistema, mas também revelaram pontos de fragilidade na lógica de determinadas funções e na organização estrutural do código. Cada erro identificado representou uma oportunidade de refinamento, permitindo à equipe reavaliar decisões anteriores, corrigir inconsistências e aprimorar a clareza e eficiência do código-fonte. Esse ciclo contínuo de teste e melhoria foi essencial para fortalecer a robustez da aplicação, bem como para otimizar sua usabilidade.

Um exemplo desse processo foi no desenvolvimento da funcionalidade *Recomendação de Hábitos*, que deu um pouco de trabalho em seu desenvolvimento. Segue um trecho do código do método que corresponde a essa funcionalidade (chamamos "função" de "método" no paradigma da orientação a objeto, por pertencer a uma classe):

```

class Recomendacao:
    def habitosSustentaveis(self):

        # Menu adaptado para futuras alterações na lista nomes_habitos_sustentaveis
        for n in list(range(len(nomes_habitos_sustentaveis))):

            # indicação de um dígito para cada recomendação da lista
            print(f'{int(n) + 1}] Adicionar o hábito {nomes_habitos_sustentaveis[n]}\n')

        # indicação do dígito para voltar
        print(f'{len(nomes_habitos_sustentaveis) + 1}] Voltar')

        escolha = str(input(Fore.YELLOW + "Escolha uma opção: ")).strip()

        # Verificação do item escolhido
        n = 0
        while n in list(range(len(nomes_habitos_sustentaveis))):
            if escolha == str(int(n) + 1):
                self.inserirHabitoRecomendacao(self, nomes_habitos_sustentaveis[n])
                break
            elif escolha == str(len(nomes_habitos_sustentaveis)):
                print(Fore.CYAN + "Voltando ao Menu de Recomendações...")
                time.sleep(1)
                break
            elif escolha == str(len(nomes_habitos_sustentaveis) + 1):
                print(Fore.RED + "Opção inválida. Tente novamente")
                time.sleep(1)
                break
            n = n + 1

```

Figure 2. Trecho do método habitosSutentaveis() da classe Recomendacao

2.2. Tecnologias Usadas (Linguagem e IDE de Desenvolvimento)

A linguagem de programação adotada no projeto foi exclusivamente o **Python 3.11**, a qual foi previamente estudada pelos membros da equipe durante o 1º período do curso de Bacharelado em Sistemas de Informação, sob orientação do professor Cleiton Magalhães — também docente responsável por este projeto na disciplina de Projetos Interdisciplinares para Sistemas de Infromação 1. A escolha do Python se deu por sua sintaxe simples e expressiva, amplamente reconhecida como ideal para o ensino de fundamentos da programação, como lógica, algoritmos e estruturas de dados. Além disso, sua vasta comunidade e ecossistema de bibliotecas tornam-no adequado tanto para aplicações educacionais quanto para projetos profissionais.

Para a persistência dos dados e armazenamento local do progresso do usuário, foi utilizada a biblioteca **SQLite3**, nativa do Python. O SQLite é um sistema de gerenciamento de banco de dados relacional (SGBD) baseado na linguagem **SQL** (Structured Query Language, ou Linguagem de Consulta Estruturada em português), um padrão amplamente adotado para a definição, manipulação e consulta de dados estruturados. Uma vez que é leve, sem necessidade de instalação de servidores adicionais e armazenar os dados em um único arquivo local, o SQLite atendeu perfeitamente às necessidades do projeto *Mutare*.

O ambiente de desenvolvimento utilizado foi a **IDE Visual Studio Code**, desenvolvida pela Microsoft. Essa ferramenta oferece uma estrutura de gerenciamento eficiente de projetos por meio de organização em pastas, recursos avançados de edição de código e suporte a extensões. Dentre essas, destaca-se a extensão *SQLite Viewer*, utilizada para inspecionar e visualizar o conteúdo do banco de dados em tempo real, facilitando os testes e o processo de depuração durante o desenvolvimento da aplicação.

2.3. Bibliotecas Usadas

bcrypt	utilizada para fazer a criptografia da senha no banco de dados
colorama	utilizada para colorir o texto exibido no terminal
datetime	utilizada para lidar com datas e horas, como registrar o momento do cadastro e login
os	limpeza de caracteres do terminal
time	utilizada para gerar delays entre as trocas de menus no terminal (time.sleep(x))
msvcrt	utilizada para capturar teclas pressionadas no terminal (Windows)
customtkinter	utilizada para gerar a interface gráfica do programa
SQLite 3	utilizada para gerar e administrar o banco de dados do sistema
Dotenv	Carregar variáveis de ambiente definidas em um arquivo .env (email remetente e senha de app para a recuperação de senha e 2FA).
MIMEMultipart	Criar e formatar o email a ser enviado.
smtplib	Enviar o email a partir do protocolo SMTP.
Random	Gerar o código.

Table 1. Bibliotecas usadas

2.4. Estrutura do Projeto

A estrutura do projeto se moldou com a ajuda de dois sistemas de repositório gratuitos: um online (virtual), o Github, e outro local, o Git; além do gerenciador de repositório Github Desktop. A equipe se formou nessas tecnologias a partir do curso gratuito disponibilizado no Youtube no canal Curso em Vídeo, do professor Gustavo Guanabara [Guanabara 2023]. Ambos os repositórios serviram para:

1. Disponibilizarmos o projeto online, pelo site do Github, com o arquivo de texto README.md para explicar sobre o projeto;
2. Versionamento do código, ou seja, criação de versões do código onde cada atualização (chamada de commit) no repositório é salva apenas como uma alteração que ela fez no código-base e como parte de um histórico de fácil acesso e gerenciamento. Por causa disso, o desenvolvimento do código conseguiu ser mais eficiente;
3. Trabalho em conjunto da equipe, com uso de branches do Github. Essa funcionalidade ramifica o repositório, de modo que cada branch vira uma versão do repositório completo, permitindo, assim, a partição do desenvolvimento entre cada funcionalidade envolvida e entre cada membro da equipe, e junção ao final de determinada etapa do projeto. Um exemplo do uso desse artifício foi para a 2ª VA, onde houveram as branches "recomendacao", para implementação das recomendações de hábitos, e "2FA" para implementação da autenticação em dois fatores.

Para o design da interface, a equipe utilizou o site *Figma*, muito utilizado por designers profissionais e que fornece uma vasta gama de ferramentas para modelagem de tela interessante e gratuita, fornecendo setas para orientar a sequência de chamada de telas de programa e espaço para colocar referências para se basear, além de trabalho colaborativo online em tempo real. Abaixo está a imagem do escopo geral do projeto visual desenvolvido. Esse design foi importante como norteador no desenvolvimento mais preciso do código da interface pela biblioteca em python CustomTKinter.

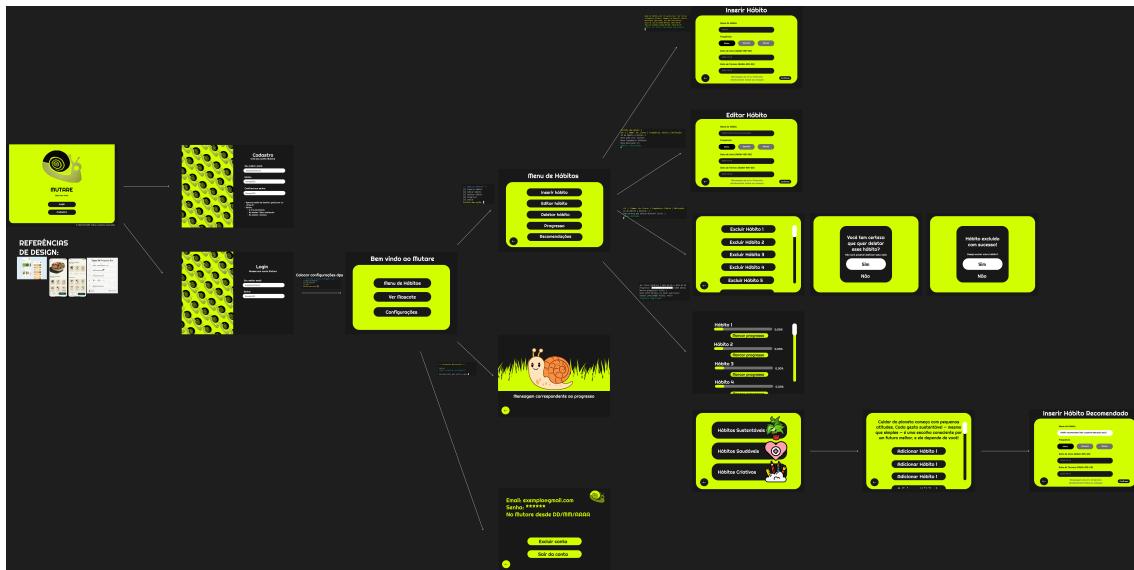


Figure 3. Design Interface do Mutare usando o Figma

Já para distribuição, acompanhamento e organização do projeto, usamos o site da ferramenta digital Trello, que funciona como uma lista de tarefas (to-do list) no estilo Kanban, que utiliza cartões arrastáveis e colunas, ambos nomeáveis, datáveis, descritivos e colaborativos. .

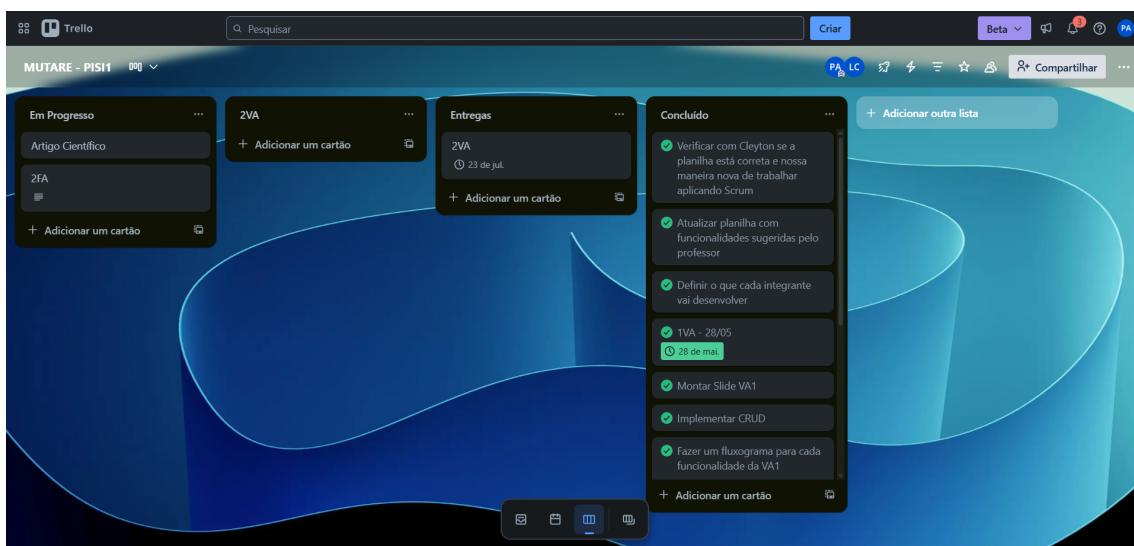


Figure 4. Print do Quadro do Trello do Mutare

3. Resultados

3.1. Release 1.0

3.1.1. Menus de Cadastro, Principal e Hábitos

Foram implementados os menus essenciais para a navegação e interação do usuário: o menu de cadastro, o menu principal e o menu de hábitos (ver Apêndice). Embora distintos em nome e função, todos compartilham uma estrutura lógica comum, descrita a seguir.

```
def menuInicial(auth):
    while True:
        Util.limparTela()
        print(Fore.MAGENTA + "\n==== MUTARE - GERENCIAMENTO DE HÁBITOS ===")
        print("[1] Login")
        print("[2] Cadastro")
        print("[3] Sair")

        opcao = input(Fore.YELLOW + "Escolha uma opção: ").strip()
        if opcao == '1':
            email = auth.loginUsuario()
            if email:
                return email
        elif opcao == '2':
            auth.cadastrarUsuario()
        elif opcao == '3':
            print("Encerrando programa...")
            time.sleep(1)
            return None
        else:
            print(Fore.RED + "Opção inválida.")
            time.sleep(1)
```

Figure 5. função *menuInicial()*, utilizada para mostrar o menu inicial.

A execução de cada menu tem início com a limpeza do terminal, realizada por meio do método *limparTela()*, garantindo uma interface mais limpa e organizada para o usuário. Em seguida, as opções disponíveis são exibidas por meio de instruções *print()*, formando o corpo visual do menu. Após a exibição, o sistema solicita ao usuário a inserção de uma entrada (*input*), armazenada na variável *opcao*, que será interpretada de acordo com as opções predefinidas para aquele contexto.

O controle de fluxo é conduzido por uma estrutura condicional (if-elif-else), em que cada ramificação corresponde a uma das opções do menu, e traz o tratamento para o caso de dígito inválido.

3.1.2. Cadastro de Usuário (C)

Essa funcionalidade funciona através do método *cadastrarUsuario()* da classe Auth.

```

def cadastrarUsuario(self):
    while True:
        Util.limparTela()
        print(Fore.WHITE + '\n==== CADASTRO DE USUÁRIO ===')
        email = input(Fore.YELLOW + 'Digite seu e-mail: ').strip()

        if not Util.emailValido(email):
            print(Fore.RED + 'E-mail inválido. Use @gmail.com ou @ufrpe.br')
            time.sleep(2)
            continue

        # Verifica se o e-mail já está cadastrado
        resultado = self.db.execute('SELECT 1 FROM usuarios WHERE Email = ?', (email,))
        if resultado.fetchone():
            print(Fore.RED + 'Este e-mail já está cadastrado. Tente outro.')
            time.sleep(2)
            continue

```

Figure 6. Print da parte 1 do método `cadastrarUsuario()`.

O método `emailValido()`, que vai ter como parâmetro o email inserido pelo usuário, vai servir para verificar se o email possui as seguintes características fundamentais para validação de um email do sistema Mutare: 1. Terminar com @gmail.com; 2. Terminar com @ufrpe.br.

a variável ”resultado” vai possuir valor caso o email seja comparado com algum do banco de dados e houver correspondência (ambos forem iguais), e isso gera uma mensagem de aviso ao usuário, e o impede de realizar o cadastro com o email inserido.

Caso ”resultado” não possua valor, significa que não há nenhum email igual ao inserido no banco de dados.

```

def cadastrarUsuario(self):
    senha = Util.inputSenhaAsteriscos('Digite a senha: ').strip()
    confirmacao = Util.inputSenhaAsteriscos('Confirme a senha: ').strip()

    validacao = Util.validarSenha(senha)
    if validacao != "válida":
        print(Fore.RED + validacao)
        time.sleep(2)
        continue

    if senha != confirmacao:
        print(Fore.RED + 'As senhas não coincidem.')
        time.sleep(2)
        continue

```

Figure 7. Print da parte 2 do método `cadastrarUsuario()`.

O método `inputSenhaAsteriscos()` da classe Util serve para mostrar asteriscos invés de dígitos, ao usuário digitar no terminal.

Em seguida duas verificações são feitas na senha e na confirmação da senha, a primeira é feita sobre a validade da senha, se ela possui uma letra maiúscula, máximo de 8 caracteres, mínimo de 4 caracteres e pelo menos um número; a segunda equipara a senha com a confirmação, a fim de garantir que a confirmação está correta.

Em seguida no código, fora do que está representado no print acima, A senha é criptografada e adicionada ao banco de dados.

3.1.3. Leitura de Usuário (R)

Essa funcionalidade foi implementada através da definição do método visualizarConta() da classe Config, e pode ser acessada através do menu de configurações do programa. É responsável por exibir as principais informações da conta do usuário no sistema Mutare, bem como oferecer opções para que ele possa atualizar sua senha ou excluir sua conta.

```
def visualizarConta(self, email, game):
    '''Exibe dados da conta e as opções de atualização/exclusão.'''
    Util.limparTela()
    conta = self.buscarConta(email)
    nivel = game.atualizarPontos()

    if not conta:
        print(Fore.RED + 'Conta não encontrada.')
        return

    print('-'*20)
    print(f"INFORMAÇÕES DA CONTA\nEmail: {conta[0]}\nSenha: {'*' * 8}\nNível: {nivel}")
    print('-'*20)

    while True:
        print('\n[1] Atualizar senha')
        print(Fore.RED + '[2] Excluir conta')
        print('[3] Voltar')
        escolha = input('Digite sua escolha: ').strip()

        if escolha == '1':
            self.atualizarSenha(email)
        elif escolha == '2':
            self.excluirConta(email)
        elif escolha == '3':
            return
        else:
            print(Fore.RED + 'Dígito inválido. Digite novamente.')
```

Figure 8. Print do método visualizarConta().

Ao ser executado, o método primeiro limpa a tela e busca os dados da conta associada ao e-mail informado. Também consulta o nível atual do usuário por meio do método atualizarPontos() da classe Gamificacao, permitindo exibir um resumo do progresso acumulado.

Em seguida, o sistema imprime na tela o e-mail do usuário, uma senha representada por asteriscos (por segurança) e o nível atual. Logo abaixo, são apresentadas três opções: atualizar a senha, excluir a conta ou voltar. Caso o usuário opte por atualizar a senha, é chamado o método `atualizarSenha()` e o método `excluirConta()`. Eles serão abordados adiante.

Essa função exemplifica bem a preocupação do sistema com a autonomia e segurança do usuário, permitindo que ele visualize suas informações com clareza, gerencie suas credenciais e, caso deseje, encerre sua participação na plataforma de forma simples e segura.

3.1.4. Atualização de Senha do Usuário (U)

O método `atualizarSenha()`, da classe `Config`, é responsável por permitir que o usuário altere sua senha de forma segura dentro do sistema `Mutare`.

```
def atualizarSenha(self, email):
    '''Atualiza a senha do usuário.'''
    conta = self.buscarConta(email)
    if not conta:
        print(Fore.RED + 'Conta não encontrada.')
        return

    senha_atual = Util.inputSenhaAsteriscos('Confirme sua senha atual: ').strip().encode('utf-8')
    senha_hash = conta[1].encode('utf-8') if isinstance(conta[1], str) else conta[1]

    if not bcrypt.checkpw(senha_atual, senha_hash):
        print(Fore.RED + 'Senha incorreta.')
        time.sleep(2)
        return

    while True:
        nova_senha = Util.inputSenhaAsteriscos('Digite sua nova senha (4-8 caracteres, ao menos uma letra maiúscula e um número): ').strip()
        if Util.validarSenha(nova_senha) == "válida":
            nova_hash = bcrypt.hashpw(nova_senha.encode('utf-8'), bcrypt.gensalt())
            self.db.execute('UPDATE usuarios SET senha = ? WHERE Email = ?', (nova_hash, email))
            self.db.conn.commit()
            print(Fore.GREEN + 'Senha atualizada com sucesso!')
            time.sleep(2)
            return
        else:
            print(Fore.RED + 'Senha inválida. Tente novamente.'
```

Figure 9. Print do método `atualizarSenha()` da classe `Auth`.

Inicialmente, ele busca os dados da conta no banco de dados com base no e-mail do usuário. Caso a conta não seja encontrada, uma mensagem de erro é exibida e o processo é interrompido. Em seguida, o método solicita que o usuário insira sua senha atual, que é digitada de forma oculta por meio da função `inputSenhaAsteriscos()`. Essa senha digitada é comparada com a senha armazenada no banco, utilizando a biblioteca `bcrypt` para garantir que a verificação aconteça de forma segura com hash criptografado, uma sequência de caracteres de tamanho fixo, tornando praticamente impossível descobrir a senha original.

Se a senha estiver correta, o sistema entra em um laço de repetição que exige que o usuário insira uma nova senha que atenda a critérios de segurança definidos: ter entre 4 e 8 caracteres, pelo menos uma letra maiúscula e um número. Esses critérios são verificados

pela função validarSenha(), do módulo Util. Caso a senha seja válida, ela é criptografada com *bcrypt* e atualizada no banco de dados.

Ao final, uma mensagem informa que a senha foi atualizada com sucesso. Se a nova senha for inválida, o usuário é orientado a tentar novamente, permanecendo no *loop* até que uma senha segura seja definida.

Essa funcionalidade é importante para que correções ou atualizações de segurança da própria conta sejam feitas pelo próprio usuário, dando-lhe ainda mais autonomia no uso do Mutare.

3.1.5. Exclusão de Usuário (D)

O método excluirConta(), da classe Config, é responsável por permitir que o próprio usuário exclua permanentemente sua conta do sistema Mutare, de forma segura e com dupla confirmação.

```
def excluirConta(self, email):
    '''Exclui a conta do usuário.'''
    conta = self.buscarConta(email)
    if not conta:
        print(Fore.RED + 'Conta não encontrada.')
        return

    senha = Util.inputSenhaAsteriscos('Confirme sua senha: ').strip().encode('utf-8')
    senha_hash = conta[1] if isinstance(conta[1], bytes) else conta[1].encode('utf-8')

    if not bcrypt.checkpw(senha, senha_hash):
        print(Fore.RED + 'Senha incorreta. Processo cancelado.')
        time.sleep(1)
        return

    confirmacao = input(Fore.YELLOW + 'Deseja mesmo excluir a conta? (s/n): ').lower()
    if confirmacao == 's':
        self.db.execute('DELETE FROM usuarios WHERE Email = ?', (email,))
        self.db.conn.commit()
        print(Fore.GREEN + 'Conta excluída com sucesso.')
        time.sleep(1)
        self.main.menuInicial(self.auth)
    else:
        print('Operação cancelada.)
```

Figure 10. Print do método excluirConta() da classe Auth.

A função começa buscando os dados da conta correspondente ao e-mail informado. Se a conta não for encontrada no banco de dados, o processo é encerrado com uma mensagem de erro.

Caso a conta exista, o sistema solicita que o usuário digite sua senha atual, de forma oculta, por meio do método inputSenhaAsteriscos(). Em seguida, a senha informada é comparada com a senha armazenada no banco de dados utilizando a função

checkpw() da biblioteca bcrypt. Já que a função checkpw() busca a senha utilizando o mesmo método feito para criptografar ela no cadastro, ele garante que a verificação ocorra de forma segura.

Se a senha estiver incorreta, a exclusão é cancelada imediatamente. Se estiver correta, o sistema ainda solicita uma confirmação textual final, perguntando se o usuário realmente deseja excluir sua conta. Apenas se o usuário confirmar com a letra “s” (sim), o comando de exclusão é executado, removendo os dados da conta da tabela de usuários. Após isso, uma mensagem de sucesso é exibida e o usuário é redirecionado para o menu inicial do sistema.

Esse método demonstra o cuidado do Mutare com a autonomia e a segurança do usuário, oferecendo uma exclusão consciente, com múltiplas camadas de verificação, evitando exclusões acidentais ou mal-intencionadas.

3.1.6. Criptografia de Senha

Para a criptografia de senha, o Mutare usa a biblioteca ”bcrypt”, que é baseada no algoritmo Blowfish adaptado para hash de senhas. Ela é caracterizada por ser adaptativa (o tempo de processamento pode ser aumentado conforme o avanço dos hardwares), lenta por design (ideal para prevenir ataques de força bruta) e segura contra ataques de rainbow tables por possuir salt embutido no hash final. O processo se inicia no método ”cadastrarUsuario()” da classe Auth, onde a senha original é convertida para bytes (.encode('utf-8')), é gerado um salt aleatório (gensalt()), o ”bcrypt.hashpw()” cria o hash final (que já inclui o salt embutido) e o hash é armazenado no banco de dados.

Em qualquer operação que exija a confirmação de senha (login, atualização, exclusão e etc), a senha digitada pelo usuário é convertida para bytes, o sistema compara com o hash armazenado no banco e o bcrypt extrai o salt automaticamente do próprio hash e refaz o cálculo para verificar se a senha confere. Na experiência final do usuário nada muda, mas com a criptografia de senha o sistema nunca vê a senha original, trazendo uma maior segurança para a aplicação.

```
try:  
    hash_senha = bcrypt.hashpw(senha.encode('utf-8'), bcrypt.gensalt()).decode('utf-8')  
    self.db.execute('INSERT INTO usuarios (Email, senha) VALUES (?, ?)', (email, hash_senha))  
    self.db.conn.commit()  
    print(Fore.GREEN + 'Usuário cadastrado com sucesso!')
```

Figure 11. Processo de criptografia no método ”cadastrarUsuario()”

3.1.7. Mascote

O método exibir(), pertencente à classe Mascote, é responsável por exibir ao usuário um mascote motivacional que reflete seu desempenho no acompanhamento dos hábitos registrados no sistema.

```

def exibir(self):
    Util.limparTela()
    print("\n==== Seu mascote motivacional ===\n")

    # Pega todos os registros de progresso
    registros = self.db.execute("SELECT id_habito FROM habito_progresso").fetchall()
    total_feitos = len(registros)

    # Calcula o total possível com base nos hábitos cadastrados
    habitos = self.db.execute("SELECT id, data_inicial, data_final, frequencia FROM habitos").fetchall()
    total_posivel = 0

    for id_habito, data_inicio, data_fim, frequencia in habitos:
        try:
            inicio = datetime.strptime(data_inicio, '%d/%m/%Y').date()
            fim = datetime.strptime(data_fim, '%d/%m/%Y').date()
        except ValueError:
            continue # ignora se as datas estiverem mal formatadas

```

Figure 12. Print da parte 1 do método `exibir()` da classe `Mascote`.

Ele começa limpando a tela e imprimindo um título introdutório, e em seguida acessa o banco de dados para contar quantos registros de progresso o usuário fez, ou seja, quantas vezes ele marcou que realizou um hábito. Depois, o método busca todos os hábitos cadastrados e, com base nas datas de início e término e na frequência de cada um (diária, semanal ou mensal), calcula quantas vezes o usuário poderia ter registrado progresso — o chamado total possível.

```

def exibir(self):
    if frequencia == 'Diária':
        total_possivel += (fim - inicio).days + 1
    elif frequencia == 'Semanal':
        total_possivel += ((fim - inicio).days // 7) + 1
    elif frequencia == 'Mensal':
        total_possivel += ((fim.year - inicio.year) * 12 + fim.month - inicio.month) + 1

    desempenho = (total_feitos / total_possivel * 100) if total_possivel > 0 else None

    # Define o estado do mascote
    if desempenho is None:
        estado = "esperando"
    elif desempenho >= 80:
        estado = "ótimo"
    elif desempenho >= 60:
        estado = "bom"
    elif desempenho >= 40:
        estado = "fraco"
    else:
        estado = "ruim"

    # Rosto e mensagem do mascote
    mensagens = {
        "ótimo": (r"\(^_)/", "Você é incrível!!!"),
        "bom": (r"\(^_)", "É isso aí, tá arrasando!!"),
        "fraco": (r"\(._.)", "Bora melhorar!"),
        "ruim": (r"\(T_T)", "Lembre-se do seu porquê, não desista agora."),
        "esperando": (r"\(o_o)", "Comece a registrar seu progresso!")
    }
}

```

Figure 13. Print da parte 2 do método `exibir()` da classe `Mascote`.

Com essas duas quantidades — total de feitos e total possível — é calculada a porcentagem de desempenho do usuário. Se ainda não houver hábitos registrados ou as datas forem inválidas, o sistema entende que o mascote está apenas ”esperando”.

Caso contrário, o método categoriza o desempenho em cinco faixas: ”ótimo” para 80% ou mais, ”bom” para 60% a 79%, ”fraco” para 40% a 59%, e ”ruim” para menos de 40%. Cada categoria está associada a um rosto emocional (feito com caracteres ASCII) e uma mensagem personalizada de incentivo, salva no dicionário ”mensagens”.

```

def exibir(self):

    rosto, mensagem = mensagens[estado]
    print(Fore.YELLOW + rosto)
    print(Fore.CYAN + mensagem)

    input("\nPressione Enter para voltar ao menu.")

```

Figure 14. Print da parte 2 do método `exibir()` da classe `Mascote`.

Por fim, o mascote correspondente é exibido a partir das variáveis "rosto" e "mascote" que recebem os valores da chave do dicionário correspondente à variável "estado", funcionando como um feedback lúdico e afetivo sobre o progresso do usuário. Depois de visualizar o mascote, o usuário pode voltar ao menu principal pressionando a tecla Enter, preferencialmente, já que qualquer outra tecla pode ser utilizada para voltar ao menu principal, uma vez que se trata apenas de um input que interrompe o programa esperando a entrada do usuário.

Essa funcionalidade fortalece a gamificação do sistema Mutare, gerando engajamento emocional e incentivando a continuidade da jornada de mudança de hábitos assim como utilizado em programas de aprendizado como o *Duolingo* e o *Forest*, aplicativos para aparelhos móveis de aprendizado e foco, respectivamente.

3.1.8. Algoritmo de Progresso do Usuário

A função calcularProgresso(), da classe Gamificacao, é responsável por calcular e exibir visualmente, com blocos coloridos e porcentagem, o andamento dos hábitos cadastrados pelo usuário no sistema Mutare.

```
def calcularProgresso(self):
    Util.limparTela()
    print("\n==== Progresso dos hábitos ===")

    habitos = self.db.execute("SELECT id, nome, data_inicial, data_final, frequencia FROM habitos").fetchall()
    if not habitos:
        print("Nenhum hábito encontrado.")
        time.sleep(1)
        return

    for habito in habitos:
        id_habito, nome, inicio, fim, freq = habito
        inicio = datetime.strptime(inicio, '%d/%m/%Y').date()
        fim = datetime.strptime(fim, '%d/%m/%Y').date()
        hoje = date.today()

        if freq == 'Diária':
            total = (fim - inicio).days + 1
        elif freq == 'Semanal':
            total = ((fim - inicio).days // 7) + 1
        elif freq == 'Mensal':
            total = ((fim.year - inicio.year) * 12 + fim.month - inicio.month) + 1
        else:
            print(f"Frequência desconhecida para {nome}.")
            continue
```

**Figure 15. Print da parte 1 do método calcularProgresso() da classe Gamifica-
cao.**

Ao ser chamada, ela começa limpando a tela e exibindo um título informativo. Em seguida, recupera todos os hábitos registrados no banco de dados, com informações como ID, nome, data inicial, data final e frequência. Caso não haja hábitos, o sistema informa e encerra a execução. Para cada hábito encontrado, as datas de início e término são convertidas para objetos do tipo date, possibilitando o cálculo do período total de execução com base na frequência: se diária, conta o número total de dias; se semanal, o número de semanas; se mensal, o número de meses entre as duas datas.

```

def calcularProgresso(self):
    feitos = self.db.execute(
        "SELECT COUNT(*) FROM habito_progresso WHERE id_habito = ?",
        (id_habito,)
    ).fetchone()[0]

    porcentagem = (feitos / total) * 100 if total else 0
    barra = '█' * int(porcentagem / 10) + '█' * (10 - int(porcentagem / 10))

    print(f"\n{name} ({freq}) | {inicio} a {fim}")
    print(f"Progresso: {barra} {porcentagem:.2f}% ({feitos}/{total})")

```

Figure 16. Print da parte 2 do método `calcularProgresso()` da classe `Gamificação`.

Com o número total de registros esperados, o sistema então busca no banco quantas vezes o usuário já marcou progresso para aquele hábito. Com esses dois números (realizados e previstos), é calculada a porcentagem de progresso, representada visualmente por uma barra composta por símbolos: blocos verdes indicam progresso alcançado e blocos vazios indicam progresso restante. Essa barra é exibida junto ao percentual e à quantidade de registros feitos.

```

def calcularProgresso(self):
    print(f"Progresso: {barra} {porcentagem:.2f}% ({feitos}/{total})")

    if input("Marcar progresso? (s/n): ").strip().lower() == 's':
        data = input("Data (DD/MM/AAAA), ou Enter para hoje: ").strip()
        if not data:
            data = hoje.strftime('%d/%m/%Y')

    try:
        datetime.strptime(data, '%d/%m/%Y')
        # verifica se já existe registro para esse hábito nessa data
        ja_existe = self.db.execute(
            "SELECT 1 FROM habito_progresso WHERE id_habito = ? AND data = ?",
            (id_habito, data)
        ).fetchone()

        if ja_existe:
            print(Fore.YELLOW + "Progresso já registrado para essa data.")
        else:
            self.db.execute(
                "INSERT INTO habito_progresso (id_habito, data) VALUES (?, ?)",
                (id_habito, data)
            )
            print(Fore.GREEN + "Progresso registrado!")
    except ValueError:
        print(Fore.RED + "Data inválida.")
    except Exception as e:
        print(Fore.RED + f"Erro ao registrar progresso: {e}")

    input("\nPressione Enter para continuar...")

```

Figure 17. Print da parte 3 do método `calcularProgresso()` da classe `Gamificação`.

Além disso, o sistema pergunta se o usuário deseja marcar um novo progresso para aquele hábito. Caso a resposta seja afirmativa, o usuário pode inserir uma data ou pressionar Enter para registrar a data atual. O sistema valida o formato da data e verifica se já existe um registro para aquele hábito no dia informado. Se não existir, um novo registro é salvo; caso já exista, uma mensagem avisa que aquele progresso já foi marcado.

Esse método é essencial na estratégia de gamificação do Mutare, pois transforma a execução de hábitos em uma experiência visualmente mensurável e interativa, incentivando o acompanhamento contínuo. Ele também se conecta diretamente com o sistema de pontuação e níveis, já que os dados de progresso alimentam a função `atualizarPontos()`, que calcula o nível do usuário com base nas ações registradas.

3.1.9. CRUD de Hábitos

O método `editarHabito()`, presente na classe `Habito`, é a principal função de atualização do CRUD de hábitos no Mutare. Ele permite ao usuário modificar um hábito já cadastrado, sendo um exemplo representativo da forma como o sistema realiza operações de edição sobre os dados.

```

def editarHabito(self):
    habitos = self.listarHabitos()
    if not habitos:
        print('Nenhum hábito encontrado.')
        time.sleep(1)
        return

    try:
        habit_id = int(input("ID do hábito a editar: "))
    except ValueError:
        print("ID inválido.")
        return

    dados = self.db.execute("SELECT nome, frequencia, motivacao FROM habitos WHERE id = ?", (habit_id,)).fetchone()
    if not dados:
        print("Nenhum hábito encontrado.")
        return

    novo_nome = input(f"Novo nome ({dados[0]}): ").strip() or dados[0]
    nova_freq = input(f"Nova frequência ({dados[1]}): ").strip() or dados[1]
    nova_motiv = input(f"Nova motivação ({dados[2]}): ").strip() or dados[2]

    self.db.execute('''
        UPDATE habitos SET nome = ?, frequencia = ?, motivacao = ? WHERE id = ?
    ''', (novo_nome, nova_freq, nova_motiv, habit_id))

    print(Fore.GREEN + f"Hábito {habit_id} atualizado.")
    time.sleep(2)

```

Figure 18. Print do método editarHabito() da classe Habito.

O método começa listando todos os hábitos registrados no banco, exibindo seus respectivos IDs, nomes, frequências e motivações. Caso não haja hábitos disponíveis, o processo é encerrado. Se houver hábitos, o usuário é solicitado a informar o ID daquele que deseja editar. O sistema valida se o ID inserido corresponde a um hábito existente, e em seguida exibe os dados atuais do hábito, oferecendo a possibilidade de alterá-los.

Para cada campo — nome, frequência e motivação — o sistema permite que o usuário digite uma nova informação ou pressione Enter para manter o valor original. Ao final, o método realiza a atualização dos dados no banco de dados com os novos valores fornecidos. Uma mensagem de sucesso confirma a operação.

Essa função é um exemplo completo de atualização de dados e representa bem a estrutura do CRUD no Mutare. Complementando essa estrutura, o método inserirHabito() representa a operação de criação (Create), permitindo que um novo hábito seja registrado com nome, frequência, motivação e período de execução, após passar por validações. O método listarHabitos() realiza a leitura dos dados (Read), exibindo todos os hábitos registrados com informações básicas. Já o método deletarHabito() realiza a exclusão (Delete), permitindo ao usuário remover um hábito com base em seu ID, após confirmação. Juntos, esses métodos implementam de forma funcional e interativa o gerenciamento completo dos hábitos do usuário dentro do sistema, e estão todos presente na classe Habito.

3.2. Release 2.0

3.2.1. Modularização e Orientação a Objeto

Seguindo a orientação do docente, modularizamos o nosso código para a configuração de arquivos a seguir:

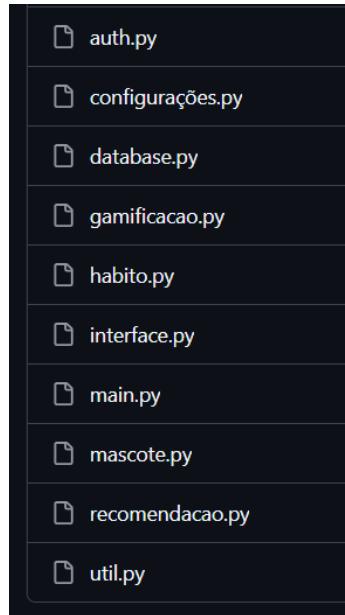


Figure 19. Arquivos em python (.py) da pasta Mutare-Project (pasta onde o está o código no repositório)

Os principais objetos do sistema são definidos nos trechos a seguir:

```
100 ˜ if __name__ == '__main__':
101      db = Database()
102      auth = Auth(db)
103 ˜     try:
104         email = menuInicial(auth)
105 ˜         if email:
106             menuPrincipal(email, db)
107 ˜     finally:
108         db.close()
```

Figure 20. Objetos db e auth presentes no arquivo *main.py* do código.

O objeto *db* foi utilizado para acessar métodos da classe *Database*, relacionada ao banco de dados (*database.db*).

O objeto *auth* foi utilizado para acessar os métodos da classe *Auth*, relacionada com cadastro, login de usuário, autenticação em dois fatores e recuperação de senha, a partir do objeto *db* para chamar métodos de banco de dados necessários nos métodos da classe *Auth*.

Além isso, a imagem representa o trecho de inicialização do código. A condição da estrutura de decisão principal *if* é que o nome do arquivo seja *main*. Essa conferência

existe para que o programa só rode o arquivo *main.py*, invés de qualquer outro programa presente em qualquer um dos arquivos importados para o main como biblioteca.

Também temos a estrutura de tratamento de erro que serve para tentar executar o programa da maneira esperada, e fechar o banco de dados (método da classe Database *db.close()*) caso ocorra algum erro

```
def menuPrincipal(email, db):
    habito = Habito(db)
    mascote = Mascote(db)
    auth = Auth(db)
    config = Config(db, main, auth)
    game = Gamificacao(db)
    rec = Recomendacao(db)
```

Figure 21. Objetos para acessar as classes do código modularizado

Cada um desses objetos acima foi utilizado para acessar uma classe diferente, presente em um dos arquivos, no formato .py, do projeto.

Todos utilizam o objeto *db* como parâmetro, uma vez que ele está ligado à classe Database, que liga o usuário à sua base de dados SQLite local (o arquivo gerado pelo código: *database.db*). Para o caso do objeto config, foram necessários também os parâmetros *main* e *auth*, garantindo que os métodos da classe *Config* pudessem usar o primeiro para chamar o Menu Inicial presente na função *menuInicial()* do arquivo *main.py* e o segundo para retornar o objeto *self.auth* como parâmetro da função *menuInicial()*, que precisa dela para executar os métodos da classe Auth que envolvem cadastro e login do usuário.

3.2.2. Autenticação em Dois Fatores Por E-mail

A autenticação em dois fatores implementada no código é um mecanismo de segurança que adiciona uma camada extra de proteção ao processo de login. Em vez de apenas exigir um e-mail e uma senha, ela também solicita um código temporário enviado por e-mail, garantindo que apenas o dono da conta (com acesso ao e-mail) consiga autenticar e acessar a aplicação. No Mutare, o processo se inicia com a inserção do e-mail e da senha na tela de login, momento em que o sistema busca no banco de dados o hash

da senha associada ao e-mail informado e, utilizando a biblioteca Bcrypt, realiza uma verificação segura comparando-a com a senha digitada. Se estiverem corretos, é gerado um código numérico aleatório de 6 dígitos, através da função ”random.randint(100000, 999999)”, como por exemplo: 462781. Este código é armazenado temporariamente e tem sua hora de criação registrada para controle de validade. Logo após, é enviado para o e-mail cadastrado a partir do protocolo SMTP(servidor smtp.gmail.com, porta 587). Isso é executado pela função ”enviarCodigoAutenticacao()” que: 1. Cria um e-mail com assunto e corpo informando o código; 2. email.mime.text.MIMEText para formatar e enviar a mensagem; 3. Usa credenciais da conta remetente carregadas de um arquivo .env.

Na sequência, o usuário é solicitado a digitar o código recebido possuindo 3 tentativas para inserir corretamente e a cada tentativa o sistema verifica se o código ainda é válido (dentro de 5 minutos). Se o código expirar, o usuário é perguntado se deseja gerar um novo código. Se o código inserido for:

Correto: o login é completado com sucesso e o usuário entra no sistema.

Incorreto: o usuário é informado e pode tentar novamente.

Após 3 falhas, o processo é encerrado.

```
# Autenticação em Duas Etapas
def gerarCodigo(self):
    self.codigo = str(random.randint(100000, 999999))
    self.hora_codigo = datetime.now()
    return self.codigo

def codigoExpirado(self):
    if not self.hora_codigo:
        return True
    return datetime.now() > self.hora_codigo + timedelta(minutes=5)

def enviarCodigoAutenticacao(self, destinatario, codigo):
    corpo = f'Seja bem vindo(a) à sua jornada Mutare! Seu código de verificação é: {codigo}'
    msg = MIMEText(corpo)
    msg['Subject'] = 'Código de Autenticação em Dois Fatores - Mutare'
    msg['From'] = self.email_remetente
    msg['To'] = destinatario

    try:
        with smtplib.SMTP('smtp.gmail.com', 587) as servidor:
            servidor.ehlo()
            servidor.starttls()
            servidor.login(self.email_remetente, self.email_senha)
            servidor.send_message(msg)
        print('✅ Código enviado com sucesso.')
        return True
    except Exception as erro:
        print(f'🔴 Erro ao enviar e-mail: {erro}')
        time.sleep(2)
        return False
```

Figure 22. Print das funções relacionadas a autenticação em dois fatores.

3.2.3. Recuperar Senha

A recuperação de senha é um processo pensado para que o usuário possa redefinir a senha de forma segura caso a esqueça, utilizando dos mesmos princípios da autenticação em dois fatores de forma que apenas o dono do e-mail cadastrado consiga trocar a senha de acesso. Durante o login, se a senha inserida não condizer com a do banco de dados, o sistema pergunta se o cliente deseja recuperar sua senha. Se a resposta for positiva, o método “recuperarSenha()” é chamado. Após isso, o programa solicita o e-mail e consulta novamente o banco de dados. Se o e-mail não estiver cadastrado, o processo é interrompido com uma mensagem:

“E-mail não encontrado.”

Caso contrário, ocorre o mesmo processo de código e verificação da funcionalidade anterior. Se as validações forem exitosas, o usuário é orientado a digitar uma nova senha, que também passa por mais um validação (presente no método Util.validarSenha()) antes de ser redefinida.

```
def recuperarSenha(self):
    Util.limparTela()
    print(Fore.WHITE + '\n==== RECUPERAÇÃO DE SENHA ===')
    email = input(Fore.YELLOW + 'Digite seu e-mail: ').strip()

    # Verifica se o e-mail está cadastrado
    resultado = self.db.execute('SELECT 1 FROM usuarios WHERE Email = ?', (email,)).fetchone()
    if not resultado:
        print(Fore.RED + 'E-mail não encontrado.')
        time.sleep(2)
        return

    # Gera e envia código
    codigo = self.gerarCodigo()
    if not self.enviarCodigoAutenticacao(email, codigo):
        print(Fore.RED + 'Falha ao enviar código. Tente novamente.')
        time.sleep(2)
        return

    print(Fore.GREEN + '✅ Código enviado. Verifique seu e-mail.')

    # Verificação do código
    for _ in range(3):
        digitado = input(Fore.YELLOW + 'Digite o código recebido: ').strip()
        if self.codigoExpirado():
            print(Fore.RED + '🔴 Código expirado. Tente novamente.')
            return
        if digitado == self.codigo:
            break
        else:
            print(Fore.RED + 'Código incorreto.')
    else:
        print(Fore.RED + '✗ Limite de tentativas atingido.')
        return

    # Redefinir senha
    nova = Util.inputSenhaAsteriscos('Nova senha(De 4-8 caracteres, pelo menos 1 número e 1 letra maiúscula): ').strip()
    confirmar = Util.inputSenhaAsteriscos('Confirme a nova senha: ').strip()

    if nova != confirmar:
        print(Fore.RED + 'As senhas não coincidem.')
        return

    validacao = Util.validarSenha(nova)
    if validacao != "válida":
        print(Fore.RED + validacao)
        return

    nova_hash = bcrypt.hashpw(nova.encode('utf-8'), bcrypt.gensalt()).decode('utf-8')
    self.db.execute('UPDATE usuarios SET senha = ? WHERE Email = ?', (nova_hash, email))
    self.db.conn.commit()

    print(Fore.GREEN + '✅ Senha atualizada com sucesso!')
    time.sleep(2)
```

Figure 23. Print da função ”recuperarSenha()”

3.2.4. Recomendação de Hábitos

A funcionalidade vai abranger toda a classe chamada Recomendacao, presente no arquivo recomendacao.py, do projeto. Ela é útil para apresentar uma lista de hábitos interessantes predefinidos para o usuário e permitir que ele adicione um desses hábitos ao seu perfil com base em preferências pessoais.

O primeiro método que é chamado (pelo menu de hábitos) dessa classe é mostrarRecomendacao(), presente no print a seguir:

```
def mostrarRecomendacao(self):
    while True:
        Util.limparTela()
        print(Fore.CYAN + f"\n==== HÁBITOS RECOMENDADOS ===")
        print("[1] Hábitos Sustentáveis")
        print("[2] Hábitos Saudáveis")
        print("[3] Hábitos Criativos")
        print("[4] Voltar")

        escolha = input(Fore.YELLOW + "Escolha uma opção: ").strip()
        if escolha == '1':
            self.habitosSustentaveis()
        elif escolha == '2':
            self.habitosSaudaveis()
        elif escolha == '3':
            self.habitosCriativos()
        elif escolha == '4':
            print("Voltando ao menu de hábitos...")
            time.sleep(1)
            break
        else:
            print(Fore.RED + "Opção inválida. Tente novamente")
            time.sleep(1)
```

Figure 24. Print do método mostrarRecomendacao() da classe Recomendacao.

Esse método é mais um menu, que vai direcionar o usuário para as opções de voltar para o menu de hábitos, tratar o erro de digitação do usuário ou seguir para cada categoria de hábitos recomendados.

A partir das 3 primeiras opções do menu, o usuário vai poder escolher um dos hábitos salvos como elemento do tipo string na lista "nomes" presente no começo de cada um dos 3 métodos (habitosSustentaveis, habitosSaudaveis e habitosCriativos), como é possível ver a seguir, nas figuras 23, 24 e 25:

```
def habitosSustentaveis(self):
    nomes = ['Comprar uma planta para cuidar', 'Reducir tempo de banho', 'Evitar uso de copos descartáveis']
```

Figure 25. Print do método habitosSustentaveis() da classe Recomendacao.

```
def habitosSaudaveis(self):
    nomes = ['Dormir no mínimo 7 horas na noite anterior', 'Beber ao menos 2 litros de água', 'Caminhar']
```

Figure 26. Print do método habitosSaudaveis() da classe Recomendacao.

```
def habitosCriativos(self):
    nomes = ['Desenhar', 'Tocar violão', 'Ler um livro', 'Sair sozinho', 'Escrever em um diário']
```

Figure 27. Print do método habitosCriativos() da classe Recomendacao.

Mais adiante, o código foi adaptado para receber quantos elementos forem colocados nessa lista, não havendo necessidade de alterar a próxima parte em uma futura atualização.

```
def habitosSaudaveis(self):
    nomes = ['Dormir no mínimo 7 horas na noite anterior', 'Beber ao menos 2 litros de água', 'Caminhar']
    while True:
        Util.limparTela()
        print('Seu corpo é sua base e sua mente é seu motor.\n')

        for i, nome in enumerate(nomes, 1):
            print(f'{i}º Adicionar o hábito {nome}\n')
        print(f'{len(nomes) + 1}º Voltar')

        escolha = input(Fore.YELLOW + "Escolha uma opção: ").strip()
        if escolha.isdigit() and 1 <= int(escolha) <= len(nomes):
            self.inserirHabitoRecomendacao(nomes[int(escolha) - 1])
            break
        elif escolha == str(len(nomes) + 1):
            print(Fore.CYAN + "Voltando ao Menu de Recomendações...")
            time.sleep(1)
            break
        else:
            print(Fore.RED + "Opção inválida. Tente novamente")
            time.sleep(1)
```

Figure 28. Print do método mostrarRecomendacao() da classe Recomendacao.

Ao ser executado, o método limpa a tela e exibe uma mensagem motivacional relacionada à importância do autocuidado.

Em seguida, são listadas três opções de hábitos saudáveis: ”Dormir no mínimo 7 horas na noite anterior”, ”Beber ao menos 2 litros de água” e ”Caminhar”, além de uma opção para voltar ao menu anterior. O usuário deve digitar o número correspondente ao hábito que deseja adicionar.

Caso escolha um dos hábitos, o método inserirHabitoRecomendacao() é chamado, abrindo um fluxo para que o usuário forneça informações complementares como frequência, motivação e datas, antes do hábito ser registrado no banco de dados. Se a opção for voltar, o método encerra o loop e retorna ao menu de recomendações. Entradas inválidas são tratadas com uma mensagem de erro e nova solicitação de escolha.

```

def inserirHabitoRecomendacao(self, habito_recomendado):
    Util.limparTela()

    self.nome = habito_recomendado
    print(f'Nome: {self.nome}')

    self.frequencia = input('Frequência (Diária, Semanal ou Mensal): ').strip().capitalize()
    if self.frequencia not in ['Diária', 'Semanal', 'Mensal']:
        print(Fore.RED + 'Frequência inválida. Tente novamente')
        time.sleep(2)
        return

    self.motivacao = input('Motivação (opcional, até 200 caracteres): ').strip()
    if self.motivacao and len(self.motivacao) > 200:
        print(Fore.RED + 'Motivação muito longa.')
        time.sleep(2)
        return

```

Figure 29. Print da parte 1 do método inserirHabitoRecomendacao() da classe Recomendacao.

Já o método inserirHabitoRecomendacao(), pertencente à classe Recomendacao, é responsável por registrar no banco de dados um hábito sugerido previamente ao usuário. Ele é chamado a partir da escolha de um hábito recomendado nas categorias disponíveis (como hábitos saudáveis, sustentáveis ou criativos).

Ao ser executado, o método limpa a tela e exibe o nome do hábito selecionado. Em seguida, solicita ao usuário que informe a frequência do hábito — podendo ser diária, semanal ou mensal — e valida essa entrada. Caso a frequência esteja fora dessas opções, uma mensagem de erro é exibida e o processo é interrompido.

Depois, o usuário pode adicionar uma motivação opcional com até 200 caracteres; caso ultrapasse esse limite, o hábito também não é registrado.

```

def inserirHabitoRecomendacao(self, habito_recomendado):
    try:
        start_date = input('Data de início (DD/MM/AAAA): ').strip()
        end_date = input('Data de término (DD/MM/AAAA): ').strip()

        start_date_dt = datetime.strptime(start_date, '%d/%m/%Y')
        end_date_dt = datetime.strptime(end_date, '%d/%m/%Y')

        if end_date_dt < start_date_dt:
            print(Fore.RED + 'Data de término anterior à de início.')
            time.sleep(2)
            return

    except ValueError:
        print(Fore.RED + 'Data inválida.')
        time.sleep(2)
        return

    try:
        self.db.execute('''
            INSERT INTO habitos (nome, criado_em, data_inicial, data_final, frequencia, motivacao, email)
            VALUES (?, ?, ?, ?, ?, ?, ?)
        ''', (self.nome, date.today(), start_date, end_date, self.frequencia, self.motivacao, self.email))

        print(Fore.GREEN + f'Hábito recomendado '{self.nome}' adicionado com sucesso!')
        time.sleep(2)
    
```

Figure 30. Print da parte 2 do método inserirHabitoRecomendacao() da classe Recomendacao.

O sistema então solicita as datas de início e término do hábito, realizando uma verificação para garantir que ambas estejam no formato correto e que a data final não seja anterior à inicial.

Se todas as entradas forem válidas, o hábito é inserido na tabela habitos do banco de dados, contendo informações como nome, data de criação, intervalo de execução, frequência, motivação e o e-mail do usuário.

Ao final, uma mensagem confirma que o hábito foi adicionado com sucesso. Caso ocorra qualquer exceção durante o processo de inserção, uma mensagem de erro é exibida.

O método é um exemplo de interação estruturada entre sistema e usuário, garantindo que hábitos recomendados sejam inseridos com segurança e com validações completas, reforçando a confiabilidade do Mutare.

3.2.5. Sistema de Pontos e Níveis

```
def atualizarPontos(self):
    Algoritmo de xp:
        Preenchimento único de hábito (diário, semanal,mensal): 1 ponto, 2 pontos e 3 pontos,
        respectivamente;
        Meta de feitos: de 20 em 20 (20, 40, 60, ...): 5 pontos;
        Para subir de nível: 5 pontos.
    """
    totalPontos = 0
    pontos = 0
    META = 20
```

Figure 31. Print da parte 1 do método atualizarPontos() da classe Gamificacao

```
habitos = self.db.execute("SELECT id, frequencia FROM habitos").fetchall()

try:
    # Verificação de 'feitos' em cada hábito registrado no banco de dados
    for id_habito, frequencia in habitos:
        feitos = self.db.execute(
            "SELECT COUNT(*) FROM habito_progresso WHERE id_habito = ?",
            (id_habito,)
        ).fetchone()[0]

        # Registro de pontos por tipo de frequência
        if frequencia == 'Diária':
            pontos = feitos * 1
        elif frequencia == 'Semanal':
            pontos = feitos * 2
        elif frequencia == 'Mensal':
            pontos = feitos * 3
```

Figure 32. Print da parte 2 do método atualizarPontos() da classe Gamificacao

A primeira parte do método vai definir as variáveis utilizadas adiante no código, além da constante META, que vai servir como parâmetro para dar pontos extras para o usuário, como será visto a seguir.

A variável habitos vai receber uma lista de tuplas (sequência de elementos (id, frequencia) representando cada linha) advinda da execução de um SELECT na aba habitos (não confundir com a variável de mesmo nome) do banco de dados (utilizando o objeto db).

Após isso, essa lista de tuplas vai ser iterada (percorrida) pelas variáveis da estrutura de controle for: id_habito e frequencia.

Antes de dar sequência, é importante definir a variável feitos. Ela vai ser importante para quantificar o número de marcações por frequência de hábito concluído. Por exemplo, se marquei, no único hábito que adicionei no Mutare, que fiz o hábito diário 4 vezes, o número de feitos salvo no banco de dados e que vai ser atribuído a variável "feitos" será 4. E essa contagem funciona a partir de um COUNT(*) no banco de dados na coluna id_habito da aba habitoprogresso, que vai contar quantas linhas foram adicionadas, que é o mesmo que contar quantos registros de conclusão foram feitos.

Seguindo adiante, vai haver uma sequência de "if" que vai buscar atribuir os pontos com base na frequência do hábito lida pelo laço no momento, utilizando os pesos já pensados para o algoritmo: 1 ponto para um registro de frequência diária, 2 pontos para um registro de frequência semanal, 3 pontos para um registro de frequência mensal.

```
def atualizarPontos(self):

    # Contadores
    total_pontos += pontos
    total_feitos += feitos

    # Pontos por meta batida
    total_pontos += (total_feitos // META) * 5

    # Subir de nível
    novo_nivel = total_pontos // 5
    self.nivel_atual = novo_nivel

    # Caso não haja hábitos registrados, não faça nenhuma operação
except:
    pass

finally:
    return self.nivel_atual
```

Figure 33. Print da parte 3 do método atualizarPontos() da classe Gamificacao

No fim de cada ciclo do laço for apresentado acima, o somatório de pontos e de

feitos é incrementado pelos valores presentes no ciclo, após as operações.

Além disso, o total de pontos pode ser acrescido de 5 pontos para cada vez que o total de feitos passar da META (por isso usamos uma divisão inteira, a operação “//”, pois ela indica apenas a quantidade de vezes que o numerador é maior do que o denominador), que é 20.

Então, o novo_nível fica definido como a parte inteira da divisão do total de pontos pelo denominador que é a quantidade de pontos para passar de nível. Após isso, o objeto vai ter seu self.nível_atual atualizado para o novo_nível calculado anteriormente.

Por fim, o método pode retornar nada, caso ocorra algum erro, tratado pelo “except” e retorna sempre, independente do erro, pelo “finally”, o self.nível_atual atrelado ao objeto.

Quando esse método for utilizado, como visto no main.py, o objeto ao qual a classe vai ser referida será o “game”.

4. Propostas Futuras

1. Gerar um relatório inteligente e personalizado do perfil de hábitos do usuário;
2. Gerar recomendações para desenvolvimento de cada hábito, e aplicação de outros. Isso se dará a partir de uma API de Inteligência Artificial. API (Interface de Programação de Aplicações) é um conjunto de definições e protocolos que permite a comunicação entre sistemas diferentes. No contexto do Mutare, será utilizada para integrar o sistema a modelos de IA capazes de analisar os dados do usuário e oferecer sugestões personalizadas de hábitos e estratégias de melhoria comportamental;
3. Implementar um aplicativo mobile do sistema Mutare, de modo a se tornar mais acessível para mais pessoas;
4. Melhorar as funcionalidades de gamificação, como recompensas, desafios e níveis, para tornar o processo de mudança de hábitos mais motivador e engajador;
5. Criar uma área de comunidade, onde os usuários possam compartilhar progressos, desafios e trocar experiências, promovendo apoio mútuo e senso de pertencimento a um grupo ciente da importância e capacidade do programa.

Essas propostas visam evoluir o Mutare de uma plataforma de apoio individual para uma ferramenta completa e integrada de transformação de hábitos, com alto potencial de impacto na saúde mental, emocional e social dos usuários.

5. Conclusão

Esse projeto teve múltiplos propósitos e impactos relevantes. Serviu como instrumento de avaliação da matriz curricular, contribuindo diretamente para a progressão acadêmica dos desenvolvedores-autores no curso. Além disso, configurou-se como um projeto prático de portfólio e uma valiosa adição ao currículo, aspectos fundamentais para inserção e destaque no mercado de trabalho. Durante sua execução, foram aprofundados conhecimentos em linguagem Python e suas bibliotecas, ambientes de desenvolvimento integrados (IDEs), plataformas de controle de versão de código (com Git e GitHub), além da modelagem e utilização de bancos de dados em sistemas digitais. Dessa forma, o projeto

Mutare não apenas cumpriu exigências acadêmicas, mas também promoveu o desenvolvimento de competências técnicas e práticas altamente demandadas na área de tecnologia.

6. Apêndice

Links do GitHub:

Desenvolvedora Maria Laura: <https://github.com/mlcordeiro>.

Desenvolvedor Pedro Ailton: <https://github.com/pedroailton>.

Docente Responsável Cleyton Vanut: <https://github.com/cvanut>.

Repositório do Projeto: <https://github.com/pedroailton/MUTARE-Project.git>.



Figure 34. Foto do *Pitch Deck* (apresentação) da Concepção do Projeto em Sala

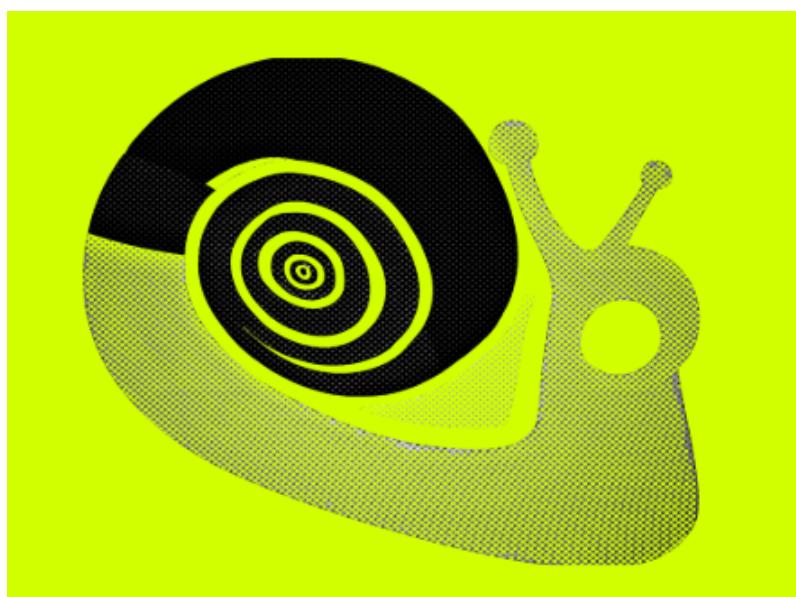


Figure 35. Foto do *Logo do programa Mutare*

2025.1 - Projetos PISI1

A1:E1 TÍTULO DO PROJETO: Mutare

ÁREA DE APLICAÇÃO: Produtividade e Gamificação

Link do projeto no Git: https://github.com/pedro.a/MUTARE-Project.git

DUPLA CODE REVIEW: Luiz Vinicius e Danielly: https://docs.google.com/document/d/1WVgXyfjwvLcOOGzDQHdC9BZGKUuPmM/edit

Email para receber notificações: email institucional: pedro.a

	A	B	C	D	E	F	G	H
1	TÍTULO DO PROJETO: Mutare	ÁREA DE APLICAÇÃO: Produtividade e Gamificação						
2	Feature	Fluxos alternativos e de erros	Entregas	Status de funcionamento	Prioridade	Escopo	CODE REVIEW:	
3		RETORNO PÓS ERRO DE DÍGITO NO MENU DE CADASTRO						Comentário
4	RF001 - Menu Cadastro	Caso o usuário digite um número que não foi prescrito no menu, ele é notificado e é solicitado que digite novamente.	30/05/2025	Pronta	P1 - Al...	1 VA		
5		VALIDAÇÃO DE EMAIL						
6		1. verificar se há .com, @, não há espaços em branco, domínio válido (ufn.edu.br); 2. verificar se o e-mail informado no cadastro é diferente de qualquer outro já cadastrado no banco de dados.						
7	RF002 - Cadastro de Conta do Usuário ("C" do CRUD)	3. no caso de invalido no email, pedir para o usuário digitar novamente.	30/05/2025	Pronta	P1 - Al...	1 VA		
8		VALIDAÇÃO DE SENHA						
9		1. verificar se há de 4 a 8 caracteres e pelo menos um número; 2. verificar se há ao menos uma letra maiúscula 3. no caso de invalido na senha, pedir para o usuário digitar novamente.						
10	RF003 - Login	VALIDAÇÃO DE LOGIN						
11		Confere se o email e a senha conferem com algum email e senha salvos no banco de dados SQLite e pede para o usuário digitar novamente em caso de não correspondência						
12	RF004 - Senha Não Visível ao Digitar	CONTROLE DA LEITURA DE SENHA						
13		O sistema troca os caracteres da senha pelo caractere * apenas na visualização da senha, entretanto é armazenada. Não afeta a usabilidade pois nenhuma senha pode ser composta só de * (asteriscos)						
14	RF006 - Menu Principal	INDICAÇÃO E PEDIDO DE REDIGITO						
		Caso o usuário digite um número que não foi prescrito no menu, ele é notificado e é solicitado que digite novamente	30/05/2025	Pronta	P1 - Al...	1 VA		

Figure 36. Trecho da planilha do Mutare no arquivo de Planilhas do Google compartilhada pelo docente Cleyton Vanut para acompanhamento dos projetos dos discentes desenvolvedores da disciplina PISI1 do período 2025.1

```
def menuPrincipal(email, db):
    mascote = Mascote(db)
    auth = Auth(db)
    config = Config(db, main, auth)
    game = Gamificacao(db)
    rec = Recomendacao(db)

    while True:
        Util.limparTela()
        print(Fore.CYAN + f"\n== BEM-VINDO AO MUTARE, {email}! ==")
        print("[1] Menu de Hábitos")
        print("[2] Ver Mascote")
        print("[3] Configurações")
        print("[4] Sair")

        escolha = input(Fore.YELLOW + "Escolha uma opção: ").strip()
        if escolha == '1':
            menuHabitos(email, habito, game, rec)
        elif escolha == '2':
            mascote.exibir()
        elif escolha == '3':
            config.menuConfiguracoes(email, game)
        elif escolha == '4':
            print("Saindo...")
            time.sleep(1)
            break
        else:
            print(Fore.RED + "Opção inválida.")
            time.sleep(1)
```

Figure 37. Função menuPrincipal() presente em main.py. Mostra o menu principal no terminal para o usuário. Segue a mesma lógica de funcionamento do menu inicial.

```

def menuHabitos(email, habito, game, rec):

    while True:
        Util.limparTela()
        print(Fore.BLUE + "\n== MENU DE HÁBITOS ==")
        print("[1] Inserir hábito")
        print("[2] Editar hábito")
        print("[3] Deletar hábito")
        print("[4] Progresso")
        print("[5] Hábitos Recomendados")
        print("[6] Voltar")

        opcao = input(Fore.YELLOW + "Escolha uma opção: ").strip()
        if opcao == '1':
            habito.inserirHabito(email)
        elif opcao == '2':
            habito.editarHabito()
        elif opcao == '3':
            habito.deletarHabito()
        elif opcao == '4':
            game.progresso()
        elif opcao == '5':
            rec.mostrarRecomendacao()
        elif opcao == '6':
            break
        else:
            print(Fore.RED + "Opção inválida.")
            time.sleep(1)

```

Figure 38. Função menuHabitos() presente em main.py. Mostra o menu de hábitos no terminal para o usuário. Segue a mesma lógica de funcionamento do menu inicial.

References

- Anjos, K. M. G. d. (2020). Relação entre os hábitos de sono, uso de mídias eletrônicas e atenção: um comparativo entre adolescentes da área urbana e suburbana da região metropolitana de natal/rn.
- Duhigg, C. (2012). *O Poder do Hábito: por que fazemos o que fazemos na vida e nos negócios*. Objetiva, 1st edition.
- G1 - Globo (2023). Número de brasileiros que sofrem com insônia chega a 73 milhões, segundo associação brasileira de sono. Acesso em: 17 jul. 2025.
- Guanabara, G. (2023). Curso em vídeo - playlist. Acesso em: 17 jul. 2025.