Trabalho de Gestão e Qualidade de Software

Professor: Calvetti

Butantã noturno

Grupo:

Angelo Rodrigues 824139676

Barbara Tracanella 824124152

Erick Domingues Soares 82414486

Eduardo Baptistella Gonçalves 824147595

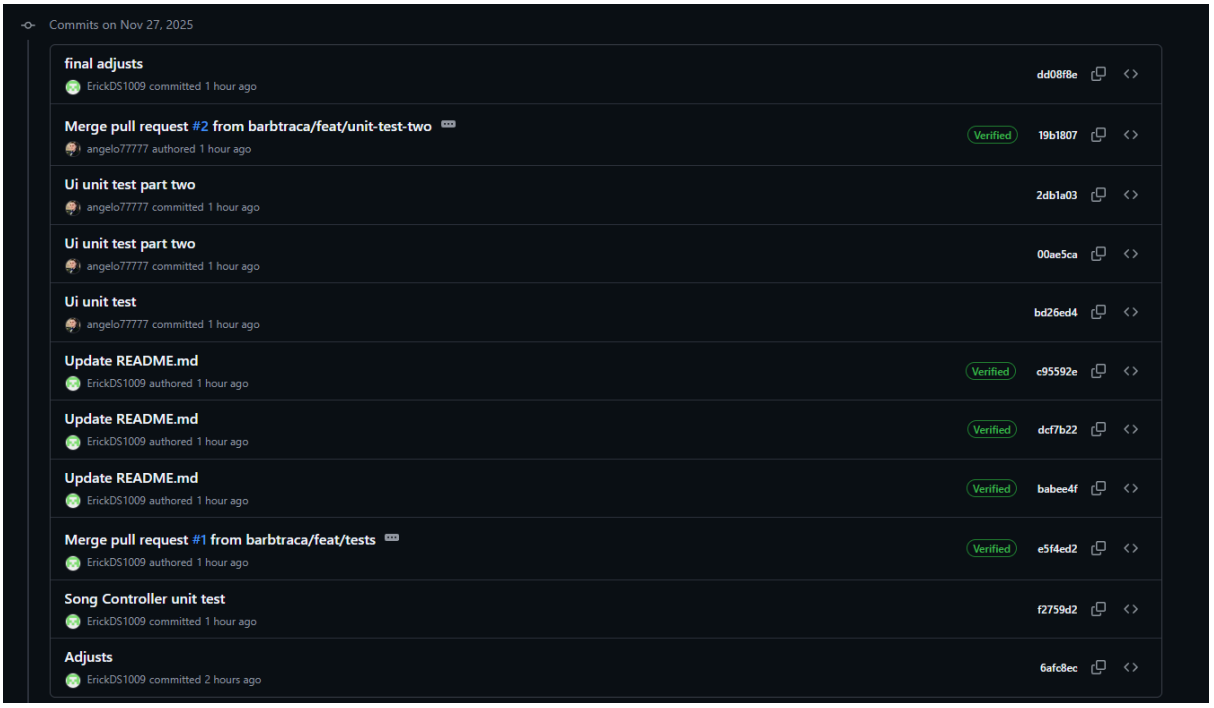
Gabriel Prieto Lima 824142064

Wellington de Oliveira Sousa 825240209

Link do Repositório GitHub Público (Código Original):

https://github.com/barbtraca/Gest-o-e-Qualidade-de-Software

Commits de todos os alunos:



1 INTRODUÇÃO E CONTEXTO DO PROJETO

O presente relatório técnico tem como objetivo documentar o processo de análise crítica, refatoração arquitetural e implementação de testes unitários para o projeto Music Player, desenvolvido em Python utilizando a biblioteca customtkinter. O estudo de caso foi conduzido sob a perspectiva da Engenharia de Software, focando na elevação da Qualidade de Código e da Manutenibilidade do sistema.

A versão inicial do Music Player, embora funcional, apresentava severas deficiências que comprometiam sua sustentabilidade a longo prazo. As principais falhas residiam no alto acoplamento entre a Interface de Usuário (UI) e suas dependências de hardware, e na ausência de documentação interna e externa. O processo de refatoração concentrou-se na aplicação de princípios de Clean Code e Isolamento de Dependência para mitigar os riscos e viabilizar a testabilidade do sistema.

2 DEFICIÊNCIAS DO CÓDIGO ORIGINAL

A análise do código-fonte original revelou problemas críticos que se manifestavam em duas áreas principais: a fragilidade arquitetural e a baixa transparência.

2.1 FRAGILIDADE ARQUITETURAL E BAIXO ACOPLAMENTO

A principal deficiência estava centralizada na classe App, que violava o Princípio da Responsabilidade Única (SRP). O construtor App.__init__ executava múltiplas ações que deveriam ser isoladas: inicialização da janela, criação de *widgets* visuais, carregamento de imagens e acoplamento indireto com o *mixer* de áudio (pygame).

Este alto acoplamento gerou um ambiente intratável para testes unitários. A tentativa de criar uma instância da App em um ambiente de teste padrão resultava na falha imediata do *thread* gráfico, manifestada pelo erro _tkinter.TclError: image "pyimageX" doesn't exist. Este erro técnico, oriundo de uma dependência de *hardware* e *thread* gráfico, demonstrou que o design do código estava incorreto, pois impedia o Teste Unitário Isolado. A dependência direta de recursos externos, como o caminho físico de arquivos de imagem, tornava o código não-determinístico e sujeito a falhas de ambiente.

2.2 DEFICIÊNCIAS DE DOCUMENTAÇÃO E TRANSPARÊNCIA

A segunda grande deficiência consistia na ausência quase total de documentação, o que comprometia a colaboração e o entendimento do código.

Em primeiro lugar, o código-fonte estava desprovido de comentários estratégicos e *docstrings*. Classes, métodos e funções careciam de explicações sobre seus objetivos, parâmetros e lógica interna. Tal ausência elevava o custo cognitivo para qualquer desenvolvedor que necessitasse dar manutenção, depurar ou entender o fluxo do programa, forçando a leitura linha a linha de todo o código.

Em segundo lugar, a ausência de um arquivo README.md completo dificultava a adoção e a execução do projeto por parte de novos colaboradores. Um

README incompleto representa a primeira barreira para a manutenibilidade, pois não fornece as instruções essenciais de pré-requisitos e instalação.

A combinação de acoplamento arquitetural e falta de documentação configurou um cenário de alto débito técnico.

3 JUSTIFICATIVAS PARA AS MUDANÇAS (REFATORAÇÃO)

A reforma implementada no projeto não foi uma simples atualização, mas sim uma medida emergencial e estratégica para resgatar o software de um estado insustentável de alto débito técnico. O código original era classificado como "ruim" por violar sistematicamente os pilares da qualidade, necessitando de intervenção imediata para garantir sua longevidade e manutenibilidade. A seguir, detalha-se por que a reforma foi essencial e como os princípios de *Clean Code* foram aplicados para reverter o cenário.

3.1 NECESSIDADE DA REFORMA: O CÓDIGO INVIÁVEL

A reforma se tornou obrigatória devido a duas falhas críticas. A primeira, de natureza arquitetural, era o Acoplamento Extremo que impedia a Testabilidade. A vinculação direta da App às dependências gráficas resultava na falha categórica (_tkinter.TclError) em qualquer ambiente de teste isolado. A segunda, de natureza colaborativa, era a Ausência de Transparência, onde o código estava desprovido de documentação interna (*docstrings*) e o README.md se encontrava incompleto ou incorreto. Esta falha comprometia a Legibilidade e inviabilizava a colaboração, elevando o custo de *onboarding* e manutenção para níveis inaceitáveis.

3.2 ESTRATÉGIA DE ISOLAMENTO E MOCKING

Para resolver a inviabilidade de testes, foi aplicada uma estratégia rigorosa de Isolamento de Dependência, garantindo a aderência ao Princípio da Inversão de Dependência (DIP).

- Patching do Construtor: O decorador @patch('src.ui.App.__init__', return_value=None) foi utilizado para substituir o construtor da App. Esta ação crucial eliminou o TclError, corrigindo a falha arquitetural e permitindo a criação de instâncias da UI em um ambiente de teste controlado.
- Isolamento de Widgets: Todas as classes de UI (CTkLabel, CTkButton) foram substituídas por objetos simulados (MagicMock). Desta forma, os testes verificam o Contrato da UI (se a função configure foi chamada) e não a renderização gráfica real, garantindo que a suíte de testes seja rápida e determinística.

- Desacoplamento do Controlador de Áudio: A instância global de song_control foi substituída por um *mock*. Isso garante o Teste de Contrato, verificando se a App chamou a lógica de negócios correta (next_song, button_action) no momento certo, isolando a responsabilidade da UI de qualquer dependência de *hardware*.
- Resolução de Interface de Teste: O ajuste na assinatura do método setUp para def setUp(self, *args): foi realizado para garantir que os *mocks* injetados pelos decoradores de classe fossem aceitos, estabilizando a suíte de testes contra erros de interface.

3.3 ADIÇÃO DE DOCUMENTAÇÃO E CLAREZA

A inclusão da documentação foi essencial para resgatar a Legibilidade e a Consistência do projeto. Foi realizada a inclusão de strings de documentação (*docstrings*) completas para classes e métodos, transformando o código em um artefato autoexplicativo. Além disso, comentários estratégicos foram adicionados para explicar a lógica complexa. Esta medida visa aumentar a transparência e reduzir o custo cognitivo, facilitando o entendimento do código por qualquer membro da equipe, o que era impossível no estado inicial do projeto.

4 DESCRIÇÃO DOS TESTES UNITÁRIOS IMPLEMENTADOS

A suíte de testes do projeto Music Player é o principal artefato que comprova a Testabilidade do código após a refatoração. A separação dos testes em módulos distintos reflete diretamente a Modularidade e o baixo acoplamento alcançados entre a UI e a Lógica de Negócios.

4.1 ESTRUTURA E DETALHAMENTO DOS TESTES UNITÁRIOS

Os testes foram categorizados e implementados em dois arquivos principais, que usam o Mocking para isolar as responsabilidades de cada camada.

4.1.1 Testes da Lógica de Negócios (testes/test_song_controller.py)

Estes testes focam exclusivamente na classe Songs, validando a lógica interna de estado, índice de reprodução e interação com o *mixer* de áudio, que é simulado via @patch.

- **Validação da Inicialização e Estado:** Testes como test_initial_state confirmam que as variáveis de controle (índice 0, *playing* False) são configuradas corretamente. O **Controle de Reprodução** (test_play_song_load_and_play, test_play_song_resume) valida o fluxo de controle, garantindo que o módulo carrega a música se estiver parada, e apenas despausa se estiver em pausa.
- **Controle de Índice e *Loop*:** Os testes garantem a gestão correta do índice de faixas, incluindo os *edge cases*. Por exemplo, test next song loop_to_start verifica que, após a última música, o índice retorna para a primeira.
- **Gestão de Recursos:** Testes como test_next_song_stops_mixer_if_playing validam que o módulo descarrega (unload) o *mixer* quando uma navegação é solicitada, garantindo a gestão correta dos recursos de áudio. O teste test_set_mixer_volume assegura a conversão correta do valor do *slider* (0-100) para o valor do *mixer* (0.0-1.0).

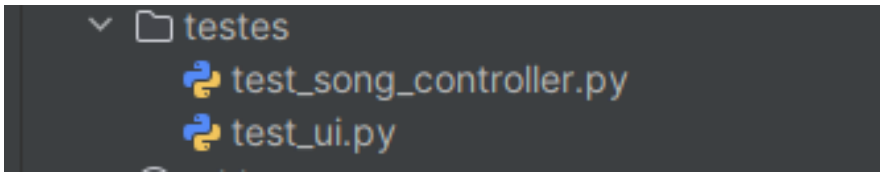
4.1.2 Testes da Interface de Usuário (testes/test_ui.py)

Estes testes focam exclusivamente na classe **App**, validando a lógica de orquestração de eventos e *feedback* visual, enquanto o controlador e todos os *widgets* são simulados via Mocking.

- **Testes de Reprodução e Estado (UI):** O teste test_toggle_play_calls_controller_and_updates_label valida que o clique no botão delega a lógica de Play/Pause ao controlador e, simultaneamente, atualiza o *label* da música. O test_toggle_play_updates_progressbar_speed_when_starting verifica se a *progress bar* visual é ativada corretamente na transição de estado. O test_toggle_play_handles_no_songs testa a robustez, garantindo que a aplicação não tenta chamar o controlador se a lista de músicas estiver vazia.

- **Testes de Navegação e Utilidade (UI):** Os testes test_handle_next_song e test_handle_previous_song validam que a UI chama o método de navegação correto no controlador e força o estado de *Play* (is_navigation=True). O teste test_set_volume assegura que o valor do *slider* é repassado corretamente ao set_mixer_volume.

- **Testes de Eventos e Encerramento:** O teste **test_handle_track_end** verifica a resposta ao evento de fim de faixa, garantindo que o controlador chama reset_variables e que o *label* exibe a mensagem de conclusão. Os testes de encerramento (test_stop_player_calls_stop_if_playing e test_stop_player_does_not_call_stop_if_not_playing) garantem que o desligamento do *mixer* é seguro e eficiente.



Link do Repositório GitHub Público (Código Original):
https://github.com/barbtraca/Gest-o-e-Qualidade-de-Software

5 conclusão código após refatoração

O código-fonte final demonstra uma melhoria em todos os aspectos de qualidade, revertendo o cenário inicial de débito técnico.

5.1. Modularidade (Responsabilidades Bem Divididas)

O código final atingiu a Modularidade completa ao aplicar o SRP e dividir o sistema em módulos autônomos: src/song_controller.py tornou-se responsável exclusivamente pela Lógica de Negócios e Áudio, enquanto src/ui.py ficou com a Apresentação e Orquestração. Essa separação garante o baixo acoplamento, permitindo que uma camada seja modificada sem quebrar a outra.

5.2. Simplicidade (O Mais Simples Possível)

A Simplicidade foi alcançada ao remover a lógica de decisão da UI. O código agora se concentra em funções que fazem apenas uma coisa. Por exemplo, métodos como toggle_play() na App são agora simples *handlers* que apenas delegam a ação principal ao controlador, sem conter lógica complexa de *mixer*.

5.3. Consistência (Padronização em Todo o Projeto)

O projeto final adota padrões consistentes, incluindo o uso rigoroso de snake_case para variáveis e funções, o uso de Constantes em *UPPER_CASE* para valores mágicos, e um padrão uniforme para *handlers* de eventos: chamar o controlador, e depois atualizar o *feedback* visual.

5.4. Legibilidade (O Código Deve Ser Autoexplicativo)

O código final é autoexplicativo graças à inclusão de strings de documentação (*docstrings*) completas e ao uso de nomes de variáveis e funções intencionais (handle_next_song, set_mixer_volume). Isso eleva a qualidade do código para o nível de "prosa", onde a leitura é facilitada e o custo cognitivo é minimizado.

5.5. Testabilidade (Código Fácil de Testar e Validar)

A Testabilidade foi totalmente restaurada e é comprovada pela execução da suíte de testes. A Testabilidade foi garantida por dois módulos de testes unitários isolados:

- Testes da Lógica de Negócios (unit_tests/test_song_controller.py): Focam exclusivamente na classe Songs, validando a lógica interna de estado e gerenciamento de índices (ex: test_next_song_loop_to_start), com *mocking* apenas para o pygame.mixer.
- Testes da Interface de Usuário (unit_tests/test_ui.py): Focam exclusivamente na classe App, validando a orquestração de eventos (ex: test_toggle_play_calls_controller_and_updates_label), com *mocking* para *todos* os *widgets* e para o controlador.

6 CONCLUSÃO SOBRE A IMPORTÂNCIA DO CLEAN CODE NA MANUTENÇÃO DE SOFTWARE

A refatoração do projeto Music Player, motivada pela inviabilidade dos testes unitários, forneceu uma demonstração prática da tese central da Engenharia de Software: o Clean Code é uma estratégia fundamental de gestão de risco e custo.

A falha do código original em ser testado foi um sintoma direto da baixa qualidade arquitetural e do alto débito técnico acumulado. A dificuldade em isolar a classe App provou que a falta de Separação de Preocupações (SRP) torna o *software* frágil e caro de manter. O esforço necessário para implementar o sistema de *mocking* e *patching* foi o preço pago pela não adoção inicial do *Clean Code*.

A adoção das práticas de Clean Code e a inclusão da documentação interna são cruciais para a manutenção de *software* por vários motivos:

1. Viabilidade de Testes Confiáveis: O código limpo é sinônimo de código desacoplado, o que permite testes unitários que são rápidos e determinísticos. A suíte de testes torna-se a principal defesa contra a Regressão, permitindo que a equipe evolua o código com confiança.
2. Redução do Custo Cognitivo: A inclusão de *docstrings* e a estruturação lógica do código eliminam o tempo gasto por desenvolvedores para decifrar a intenção por trás das funções. O código se torna fácil de ler, o que é um fator chave para a produtividade e a colaboração.
3. Resiliência Arquitetural: Princípios como a Inversão de Dependência (DIP) garantem que as mudanças em uma camada (ex: a UI) não exijam reescritas ou quebras de testes em outras camadas (ex: o Controlador).

Em síntese, a reforma do Music Player provou que o investimento em *Clean Code* não é uma estética, mas sim um investimento estratégico que reduz o risco de *bugs*, acelera o desenvolvimento futuro e garante a sustentabilidade e a evolução contínua do produto de *software*.