PROJETO A3 - GESTÃO E QUALIDADE DE SOFTWARE

PRODUTO DE SOFTWARE - YOUTUBE GAMER DOWNLOADER

ORIENTADOR: Robson Calvetti

Nome dos Alunos:
Gabriel Rodrigues da Silva Costa - 822137024
Lucas Ribeiro Pedroso - 823149292
Thiago Duarte Reis - 822141527

São Paulo
2025

Relatório Técnico: Análise do Código Legado

Código legado

1. Objetivo do Código

1.1. O código fonte (`Código legado`) implementou um sistema simples de cadastro e login utilizando Tkinter como interface gráfica e armazenamento de usuários em um arquivo JSON.

Este código foi propositalmente construído com más práticas para fins acadêmicos, permitindo análise, identificação de problemas e aplicação de técnicas de refatoração conforme os princípios de Clean Code, SOLID e boas práticas de desenvolvimento.

O objetivo deste relatório é documentar as falhas encontradas no código legado, justificar as mudanças realizadas e apresentar a versão refatorada (`youtube_downloader_refactored`), demonstrando melhorias de organização, segurança e manutenibilidade.

Link da pasta do Código Legado:
https://github.com/LucasrPedroso/ProjetoA3--GQS/tree/main/Codigo%20legado

2. Problemas Identificados

Categoria	Problema	Consequência
Arquitetura	Interface, lógica e persistência misturadas	Código difícil de manter, testar e expandir
Boas Práticas	Código executado automaticamente ao importar o arquivo	Impede modularidade e inviabiliza testes unitários
Segurança	Senhas armazenadas com MD5 sem salt	Vulnerável a ataques e quebra fácil de hash
Dados	Persistência usando JSON diretamente	Risco de corrupção e ausência de controle de concorrência
Nomenclatura	Funções e variáveis com nomes genéricos	Reduz entendimento e clareza
Padrões de Projeto	Ausência de design pattern	Código procedural com forte acoplamento
Testabilidade	Funções dependem de interface gráfica	Testabilidade extremamente limitada

Tratamento de erros	Uso de except: genérico	Erros silenciosos e difícil depuração
POO	Código 100% procedural	Ausência de orientação a objetos e modularidade

3. Código Executado no Import

3.1. O arquivo original executava a função principal ao ser importado:

```
legacy_main_window()
```

3.2. **Problemas:**

- Gera efeitos colaterais inesperados
- Impede importação do módulo para testes e reuso

3.3. **Correção:**

```
if __name__ == "__main__":
    legacy_main_window()
```

4. Persistência de Dados

O sistema salva usuários diretamente em um arquivo JSON utilizando funções simples:

- **load_users()**
- **save_users()**
- **find_user()**
- **add_user()**

4.1. **Problemas:**

- IDs são gerados com `random.randint()`, podendo colidir
- Falta de validação de dados
- Ausência de camadas de abstração (Repository Pattern inexistente)

4.2. Correção:

- Criação de uma camada de persistência separada na refatoração (`repository.py`)
- Estrutura preparada para futuras substituições por banco de dados real

5. Segurança

O código armazenava senhas utilizando hashing MD5 sem salt.

```
hashlib.md5(password.encode()).hexdigest()
```

5.1. Problemas:

- MD5 é considerado inseguro e quebrável
- Não há uso de sal (salt)
- Não há proteção contra força bruta

5.2. Correção:

- Versão refatorada permite substituição do hash por algoritmos modernos como **bcrypt** ou **argon2**

6. Interface Gráfica (Tkinter)

A UI estava acoplada diretamente à lógica de negócio e persistência:

- Componentes não reaproveitáveis
- Dificuldade extrema de testar sem interface
- Componentes não são reutilizáveis

6.1. Correção:

- Separação em módulos na refatoração (UI separada da lógica).

7. Testabilidade

O código original era impossível de testar em nível de função devido ao acoplamento.

7.1. Correção:

- Separar lógica do UI
- Criar testes com pytest

8. Princípios Violados

Princípio	Violação
SRP (Responsabilidade Única)	Funções fazem UI + validação + persistência
OCP (Aberto/Fechado)	Alterações exigem editar funções existentes
DIP (Inversão de Dependência)	Dependência direta de implementação concreta
DRY (Don't Repeat Yourself)	Repetição de validações e mensagens
KISS	Soluções simples poderiam ser aplicadas
YAGNI	Condições extra sem necessidade real

9. Recomendações de Refatoração

- Aplicação de princípios SOLID
- Modularização da lógica
- Uso de Repository Pattern
- Melhoria de nomenclaturas e legibilidade
- Preparação para testes unitários
- Código executável somente por função principal

9.1. Arquitetura em camadas

- Ui
- Services

- Models
- Config
- Utils

9.2. Estrutura do projeto

```
youtube_downloader_refactored/
├── src/
│   ├── config/             # Configurações e constantes
│   │   ├── firebase_config.py
│   │   └── constants.py
│   ├── models/             # Modelos de dados
│   │   ├── video_info.py
│   │   └── exceptions.py
│   ├── services/           # Lógica de negócio
│   │   ├── auth_service.py
│   │   ├── download_service.py
│   │   └── video_info_service.py
│   ├── ui/                 # Interface gráfica
│   │   ├── base_components.py
│   │   ├── login_window.py
│   │   ├── downloader_window.py
│   │   └── download_thread.py
│   ├── utils/              # Utilitários
│   │   ├── validators.py
│   │   └── system_utils.py
│   └── app_controller.py   # Controlador principal
├── tests/                  # Testes unitários
│   ├── test_auth_service.py
│   ├── test_download_service.py
│   ├── test_video_info_service.py
│   └── test_validators.py
├── main.py                 # Ponto de entrada
└── requirements.txt        # Dependências
```

9.3. Requisitos

- Python 3.7+
- PyQt5
- yt-dlp
- requests
- Pyrebase4
- FFmpeg (para conversão de vídeos)

Link do Código Refatorado:
https://github.com/LucasrPedroso/ProjetoA3--GQS/tree/main/youtube_downloader_refactored

10. Descrição dos Testes Unitários Aplicados

10.1. Os testes unitários foram desenvolvidos com o objetivo de validar o comportamento da aplicação refatorada, garantindo que as funcionalidades essenciais continuem operando corretamente após as alterações estruturais realizadas no código.

A abordagem utilizada segue os princípios de **Test-Driven Development (TDD)** em alguns módulos e validações após-refatoração em outros.

10.2. Abrangência dos Testes

Os testes aplicados focaram nos seguintes aspectos do sistema:

Categoria de Teste	Objetivo	Status
Testes de Validação	Garantir que entradas inválidas sejam rejeitadas	Implementados
Testes de Lógica de Negócio	Verificar o comportamento correto das funções principais	Implementados
Testes de Persistência (simulação)	Verificar leitura e escrita de dados de forma isolada	Implementados (usando mocks)
Testes de Download (YouTube)	Validar execução do processo de download	Parcial — depende de API externa
Testes de Interface Gráfica (Tkinter)	Não aplicável a testes unitários	Não implementados

10.3. Exemplos dos Testes Criados

- Testes de validação

```
assert validate_url("https://youtube.com/watch?v=123") == True
assert validate_url("teste invalido") == False
```

10.4. Testes da lógica principal

- Garantem que a função de download execute o fluxo esperado:

```
result = downloader.download_video("https://youtube.com/example")
assert result.success is True
```

10.5. Testes de persistência (mock)

- Utilizam simulação de banco de dados para evitar dependências reais:

```
mock_repo.save("user123")
assert mock_repo.get("user123") is not None
```

10.6. Resultado dos Testes

Após a execução dos testes, os seguintes resultados foram observados:

- Todas as funções essenciais retornaram ao comportamento esperado.
- Não foram identificadas regressões conhecidas após a refatoração.
- O código apresentou melhor previsibilidade e facilidade de isolamento durante a testagem.

10.7. Localização dos Testes no Repositório

A estrutura dos arquivos segue o padrão:

```
tests
  __init__.py
  test_auth_service.py
  test_download_service.py
  test_validators.py
  test_video_info_service.py
```

Link da pasta tests:

https://github.com/LucasrPedroso/ProjetoA3--GQS/tree/main/youtube_downloader_refactored/tests

11. Conclusão sobre a Importância do Clean Code na Manutenção de Software:

A aplicação de Clean Code na refatoração demonstrou ser fundamental para tornar o sistema mais compreensível, sustentável e simples de evoluir. Um código limpo facilita a leitura, torna o entendimento mais fácil envolvido na compreensão de funcionalidades e diminui a probabilidade de erros em futuras modificações.

Além disso, boas práticas como modularização, nomes descritivos, redução de acoplamento e funções com responsabilidade única tornam o software previsível e mais fácil de testar. Isso impacta diretamente o tempo e o custo de manutenção, que representam a maior parte do ciclo de vida de um software.

Portanto, a importância do Clean Code vai além da estética ou organização, ele contribui para um desenvolvimento mais eficiente, maior qualidade técnica, menor retrabalho e maior confiabilidade do sistema ao longo do tempo. De modo geral, escrever código limpo é investir na longevidade, escalabilidade e manutenção saudável do software.

12. Link do Repositório:

https://github.com/LucasrPedroso/ProjetoA3--GQS.git

13. Demonstração Online do Sistema:

Video A3 - GQS

Caso haja erro na pasta do vídeo:
https://drive.google.com/drive/folders/1iAE4Co7Ka7ex0ufXJfhXrTQ3rj0GZB49?usp=drive_link