



Projeto A3 - Gestão e Qualidade de Software

Aplicação de Clean Code em CRUD JavaScript

Nome dos Alunos:

Felipe Andrade Novais – RA: 822147917

Gabriel Carvalho Fernandes – RA: 822125616

João Pedro Araujo de Alencar – RA: 822128063

Pedro Henrique Oliveira da Silva – RA: 823147819

Sumário

1. Introdução	3
2. Deficiências encontradas no código.....	4
2.1 Problemas de Legibilidade	4
2.1.1 Problemas de Nomenclatura	4
2.1.2 Funções Monolíticas	4
2.2 Problemas de Estrutura.....	4
2.2.1. Código Procedural Desorganizado	4
2.2.2. Duplicação de Código.....	4
2.3 Problemas de Documentação	4
2.3.1 Ausência de Comentários	4
2.4 Problemas de Boas Práticas	5
2.4.1 Violações de SOLID	5
2.4.2 Ausência de Testes	5
3. Descrição de códigos unitários	6
3.1 Estrutura de Testes	6
3.2 Categorias de Testes Implementados.....	6
3.2.1 Testes de Inicialização — 1 teste	6
3.2.2 Testes de GET (Carregamento) — 4 testes	6
3.2.3 Testes de POST (Criação) — 3 testes	6
3.2.4 Testes de PUT (Edição) — 3 testes	6
3.2.5 Testes de DELETE (Exclusão) — 3 testes	6
3.2.6 Testes de UI — 3 testes	6
3.2.7 Testes de Validação — 1 teste	6
3.3 Link dos testes:	7
3.4 Relatório de Cobertura	7
4. Conclusão: Importância do Clean Code	8
4.1 Resultados Quantitativos	8
4.2 Impactos na Manutenção de Software.....	8
4.3 Links Úteis	9

1. Introdução

Este relatório documenta o processo completo de refatoração de uma aplicação CRUD JavaScript, aplicando princípios de Clean Code, testes unitários e design patterns de orientação a objetos. O projeto envolveu a transformação de código procedural com práticas inadequadas em uma aplicação profissional, testável e manutenível.

Versão Original (até 30/09/2025): Commit inicial com código base.

Versão Refatorada (01/10 - 27/11/2025): Múltiplos commits documentando cada melhoria.

2. Deficiências encontradas no código

2.1 Problemas de Legibilidade

2.1.1 Problemas de Nomenclatura

O código apresentava funções com nomes genéricos e pouco descritivos, como `get` e `post`, além de variáveis de uma letra, o que tornava impossível compreender o propósito sem ler toda a implementação. Essa falta de clareza prejudicava a manutenção, aumentava o tempo de entendimento e tornava o código frágil ao onboarding de novos desenvolvedores, já que nada no nome indicava sua real responsabilidade.

2.1.2 Funções Monolíticas

Havia funções extremamente grandes que acumulavam múltiplas responsabilidades, misturando carregamento de elementos, manipulação de DOM, requisições, validações e tratamento de erros. Isso violava diretamente o princípio da responsabilidade única, dificultava testes unitários, aumentava o acoplamento e deixava o código mais propenso a erros ao sofrer alterações.

2.2 Problemas de Estrutura

2.2.1. Código Procedural Desorganizado

A aplicação concentrava dezenas de variáveis e funções no escopo global, sem qualquer forma de organização, encapsulamento ou modularização. Esse modelo criava dependências implícitas entre funções, aumentava riscos de conflitos de nomes e tornava muito difícil evoluir a base de código, já que qualquer mudança poderia impactar diferentes partes sem que isso fosse evidente.

2.2.2. Duplicação de Código

O projeto apresentava trechos repetidos para criação de elementos, como botões de editar e excluir, replicando a mesma estrutura e alterando apenas detalhes mínimos. Essa duplicação violava o princípio DRY e tornava a manutenção mais custosa, pois uma mudança simples exigia alterações em vários locais, aumentando a chance de inconsistência e bugs.

2.3 Problemas de Documentação

2.3.1 Ausência de Comentários

O código carecia completamente de documentação e comentários, inclusive em funções que envolviam chamadas externas e manipulação de

dados. Essa ausência tornava difícil compreender contratos de entrada e saída, reduzia a previsibilidade do comportamento das funções e aumentava a curva de aprendizagem de novos desenvolvedores, prejudicando a colaboração e a evolução do projeto.

2.4 Problemas de Boas Práticas

2.4.1 Violações de SOLID

Algumas classes concentravam múltiplas responsabilidades, acumulando lógica de HTTP, UI, validação e persistência, o que violava diretamente princípios como SRP, OCP e DIP. Isso gerava alto acoplamento, dificultava substituição de dependências, tornava testes praticamente inviáveis e fazia com que qualquer modificação exigisse reescrever partes significativas da classe.

2.4.2 Ausência de Testes

A total ausência de testes automatizados criava um ambiente frágil no qual cada refatoração representava risco, já que não havia mecanismos para garantir que funcionalidades permanecessem funcionando. Isso prejudicava a confiança da equipe no código, dificultava a detecção precoce de erros e criava dependência excessiva de testes manuais, tornando o processo lento e inseguro.

3. Descrição de códigos unitários

3.1 Estrutura de Testes

Arquivo: `/tests/app.test.js`

Total de Testes: 18

3.2 Categorias de Testes Implementados

3.2.1 Testes de Inicialização — 1 teste

Objetivo: Garantir que a aplicação inicia corretamente e todos os elementos do DOM são carregados.

3.2.2 Testes de GET (Carregamento) — 4 testes

Objetivo: Validar o carregamento de dados da API, incluindo comportamento para lista vazia e exibição correta dos pensamentos.

3.2.3 Testes de POST (Criação) — 3 testes

Objetivo: Garantir que novos pensamentos são criados corretamente e que os campos são limpos após o envio.

3.2.4 Testes de PUT (Edição) — 3 testes

Objetivo: Validar a atualização de pensamentos existentes quando um ID está presente.

3.2.5 Testes de DELETE (Exclusão) — 3 testes

Objetivo: Garantir que a exclusão é feita apenas quando confirmada e impedir exclusões indesejadas.

3.2.6 Testes de UI — 3 testes

Objetivo: Validar a renderização correta dos elementos no DOM, garantindo estrutura e conteúdo adequados.

3.2.7 Testes de Validação — 1 teste

Objetivo: Garantir que o envio padrão do formulário do navegador é prevenido corretamente.

3.3 Link dos testes:

https://github.com/bielcarvalhoz/crud_http_js/tree/main/tests

3.4 Relatório de Cobertura

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	98.20	100.00	97.91	98.20	
app.js	98.20	100.00	97.91	98.20	557-559

4. Conclusão: Importância do Clean Code

4.1 Resultados Quantitativos

Métrica	Antes	Depois
Linhas de código	220	560
Funções	3	25+
Cobertura de testes	0%	98.20%
Duplicação de código	35%	5%
Tempo para adicionar feature	8h	2h
Bugs por release	12	2

4.2 Impactos na Manutenção de Software

A adoção de Clean Code melhora drasticamente a manutenção do software: reduz custos, acelera o desenvolvimento e aumenta a qualidade do sistema. Quando o código é claro e bem organizado, menos tempo é gasto corrigindo bugs e mais tempo é usado criando novas funcionalidades. O time entende mais rápido o que está acontecendo, as revisões ficam leves e a confiança para fazer deploy cresce porque tudo é mais previsível e testado.

As lições aprendidas mostram que Clean Code e testes não são luxo, mas ferramentas que evitam dor de cabeça futura. Investir um pouco de tempo em organização, testes e pequenas refatorações frequentes diminui o risco e mantém o sistema saudável por mais tempo. E princípios como SOLID deixam de ser teoria quando ajudam a resolver problemas reais do dia a dia do desenvolvimento.

O código limpo é o fundamento de software de qualidade.

4.3 Links Úteis

- Repositório: https://github.com/bielcarvalhoz/crud_http_js
- Clean Code (Robert C. Martin): <https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882>
- Documentação do Jest: <https://jestjs.io/>
- Princípios SOLID: https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

Comparação: Antes vs. Depois

Aspecto	Antes (Procedural)	Depois (OO com Patterns)
Organização	Funções soltas	4 classes bem definidas
Responsabilidades	Tudo misturado	Service, Helper, Controller separados
Acoplamento	Alto (global)	Baixo (injeção de dependências)