Universidade Católica de Brasília - UCB

Qualidade de Software e Governança

Análise Estática e Refatoração de Código em um Projeto real

Aluno: Caio Seabra de Queiroz
Professor: Oscar Galdino de Oliveira Junior

Ceilândia - DF

2025

Sumário

1 – Descrição do Projeto Selecionado

1.1. Nome do Projeto

- Vanilla Js ToDoList

1.2. Link do Repositório Original

- Autor: Nik Krivoshei
- **Link Repositório GitHub**

1.3. Descrição

Este projeto consiste em uma aplicação web simples de lista de tarefas (To-Do List), que permite ao usuário adicionar, marcar como concluídas e remover tarefas.

1.4. Tecnologias Utilizadas

HTML, CSS E JavaScript.

1.5. Motivo da Escolha

O projeto foi escolhido por ter tamanho considerável de código, mas conter lógica de manipulação de DOM, funções com estrutura condicional e repetição, o que permite analisar complexidade e boas práticas.

2 – Análise Estática de Código

2.1. Ferramentas Utilizadas

- **ESlint**: análise de estilo e boas práticas de JavaScript.
- **SonarCloud:** avaliação geral de qualidade de código (complexidade, duplicação, vulnerabilidade, dívida técnica).

2.2. Resultados do ESLint

- Após executar o comando:

  npx eslint . --format html -o analise/relatorio-eslint.html

O relatório “relatório-eslint.html” apresentou o seguinte resultado:

- **Total de problemas encontrados:**
  - **Erros: 0**
  - **Avisos: 1 ('e' is defined but never used)**

Arquivo	**Linha**	**Tipo**	**Descrição**
App.js	52:45	Warning	‘e’ is defined but never used (variável declarada mas não utilizada).

O aviso indica que a variável **“e”** foi declarada, mas não utilizada no código — algo comum em funções de evento (por exemplo**, function handleClick(e)**), quando o parâmetro é definido, mas não é necessário dentro da função.

Esse tipo de aviso não compromete o funcionamento do código, mas pode ser corrigido removendo a variável não utilizada para manter o código mais limpo.

Figura 1- Relatório gerado pelo ESLint

2.3. Resultados do SonarCloud

A análise do SonarCloud foi executada na branch main do repositório, e forneceu uma visão detalhada sobre a qualidade do código, contemplando métricas como vulnerabilidade, confiabilidade, manutenibilidade, duplicações e complexidades ciclomática.

2.3.1. Visão Geral das Issues Identificadas

Foram identificados 14 problemas (issues) no total, distribuídos conforme os critérios abaixo:

- **Security:** 0 problemas (nenhuma vulnerabilidade crítica detectada).
- **Reliability:** 3 problemas (podem ocasionar erros de execução).
- **Maintainability (Code Smells/ Dívida Técnica):** 11 problemas.

Classificação por severidade:

- **1 Blocker** - problema extremamente crítico, deve ser corrigido imediatamente.
- **1 High** - alta prioridade.
- **3 Medium** - prioridade moderada.
- **5 Low** - baixa severidade.
- **4 Informational** - sugestões e boas práticas.

2.3.2. Complexidade Ciclomática

A métrica de complexidade ciclomática analisada pelo SonarCloud indica o nível de caminhos possíveis dentro das funções do código, ou seja, quantas decisões, condicionais ou ramificações e aplicação possui.

De acordo com a análise:

- O nível geral de complexidade do projeto é 13, sendo 12 do arquivo app.js e 1 do index.html (valor do ESLint ignorado).
- Nenhuma função ultrapassa níveis críticos de complexidade.
- O projeto possui três arquivos: index.html, style.css e app.js.

Figura 2- Detalhamento da complexidade ciclomática por arquivo

2.3.3. Cobertura de Código (Code Coverage)

A métrica de cobertura de código indica qual porcentagem da aplicação é coberta por testes automatizados (como testes unitários ou de integração).
No entanto, este projeto não possui nenhum tipo de teste automatizado implementado, o que resulta em:

- Cobertura total: 0%

Esse cenário é esperado e aceitável em projetos pequenos e simples como este, especialmente quando o objetivo principal é treinar manipulação de DOM ou interações básicas com o usuário.

Ainda assim, a ausência de testes aumenta o risco de regressões futuras, justificando a implementação de testes em uma eventual refatoração.

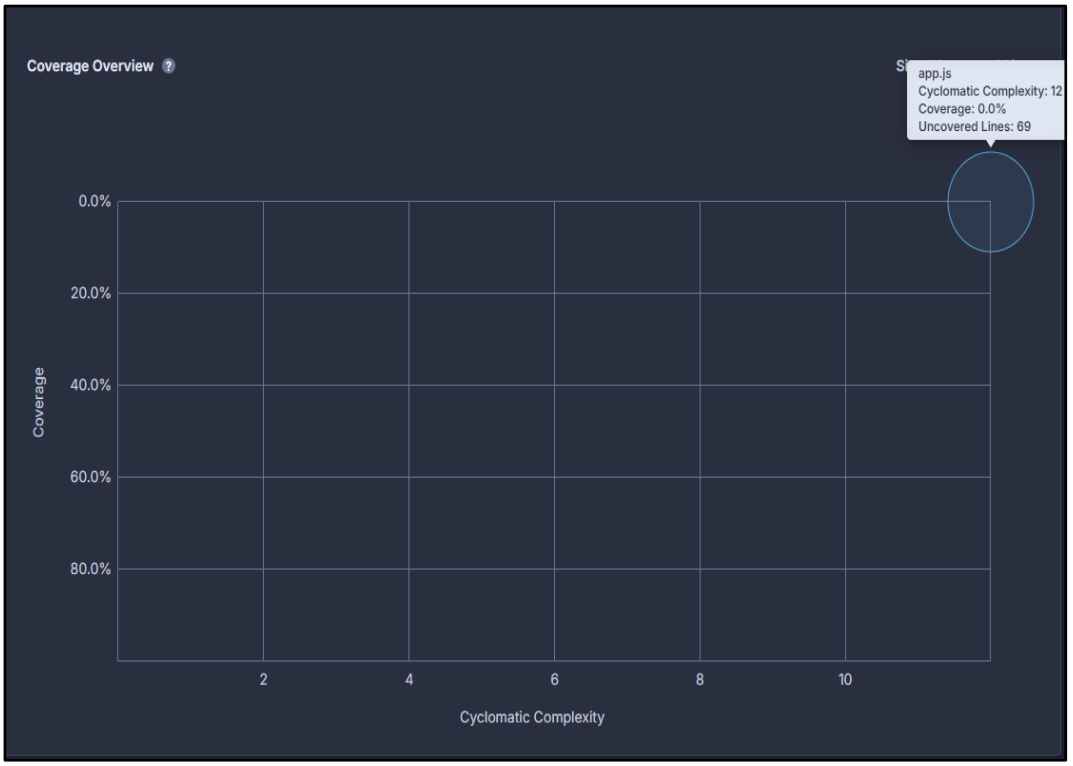

Figura 3- Cobertura de código reportada pelo SonarCloud

2.3.4. Dívida Técnico (Technical Debt)

A dívida técnica representa o tempo estimado que seria necessário para corrigir problemas de manutenibilidade no código, como más práticas, duplicação ou estruturação inadequada.

Segundo o SonarCloud, o projeto apresenta:

- Dívida técnica classificada como baixa
- O tempo estimado para correção é pequeno (minutos ou poucas horas)
- As issues encontradas estão relacionadas principalmente a boas práticas e organização de código, não a falhas funcionais

Esse resultado reforça que o projeto é simples, funcional e legível, mas pode ser otimizado para facilitar futuras manutenções e evoluções

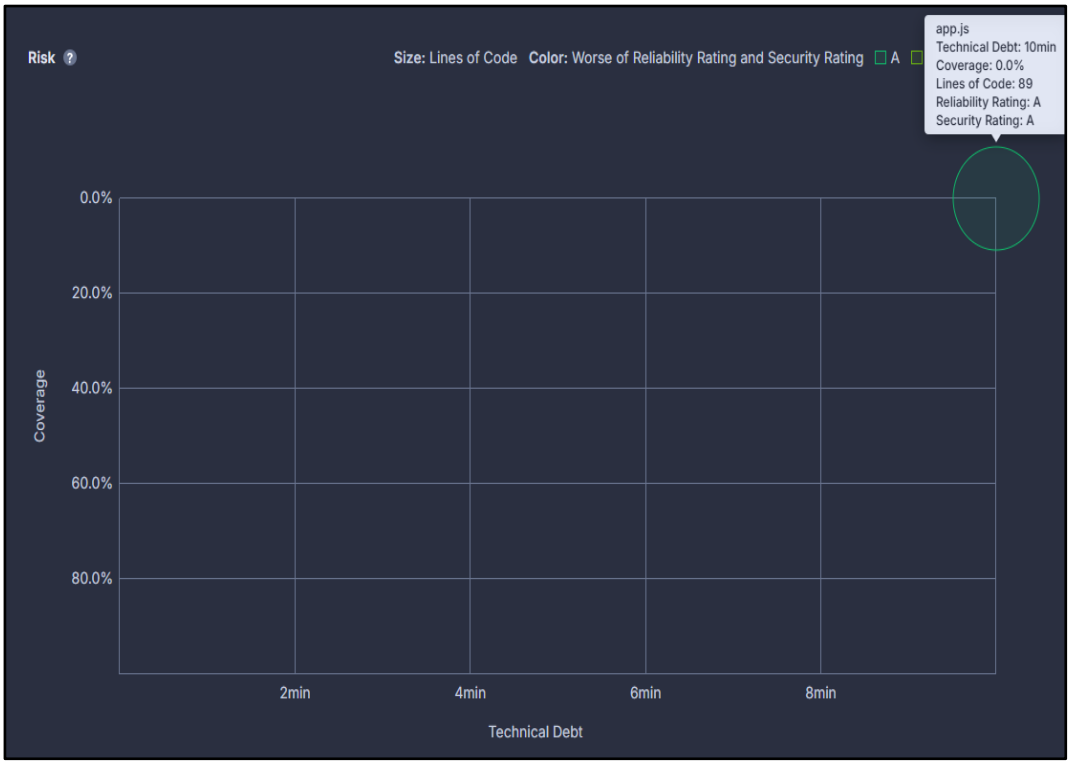

Figura 4- Estimativa de dívida técnica pelo SonarCloud

2.3.5. Código Duplicado (Duplications)

O SonarCloud também analisou o nível de código duplicado no projeto. O relatório indicou:

- Nenhum trecho relevante de duplicação de código foi identificado

Isso mostra que o desenvolvedor evitou repetição desnecessária de lógica, contribuindo positivamente para a manutenibilidade e legibilidade do projeto.

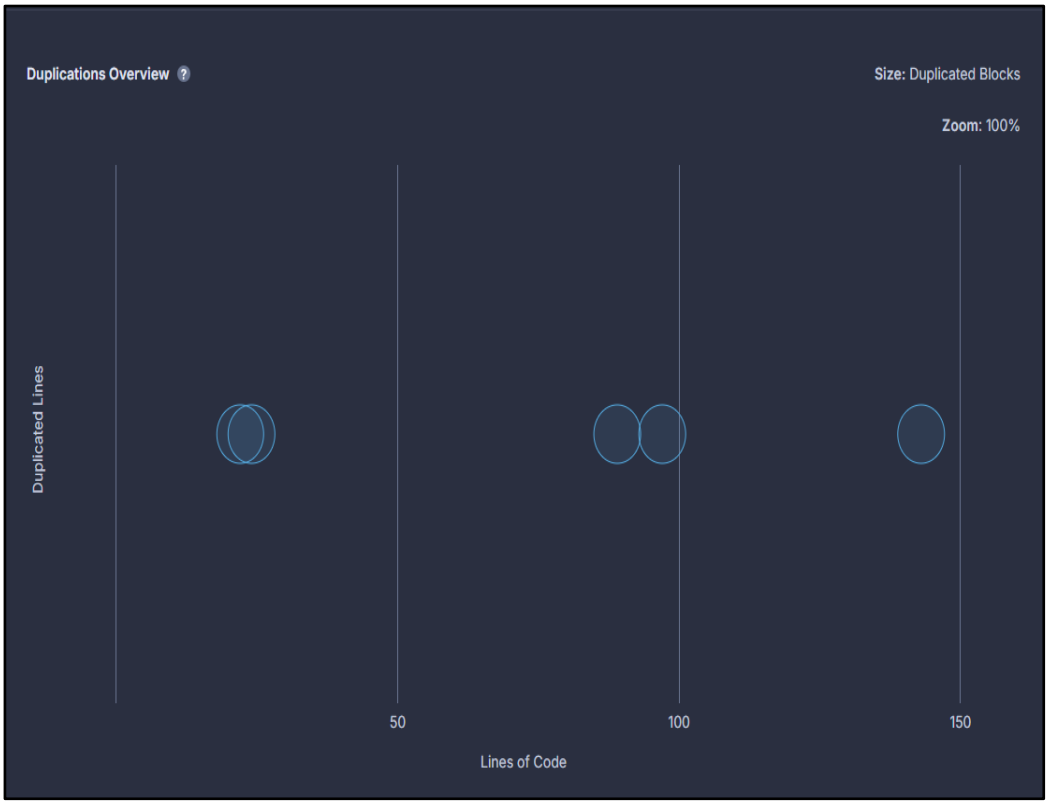

Figura 5 - Indicador de duplicação de código no SonarCloud

2.3.6. Vulnerabilidade de Segurança

A análise de segurança realizada pelo SonarCloud verificou possíveis riscos de exploração por usuários maliciosos.

Resultado encontrado:

- Nenhuma vulnerabilidade crítica de segurança identificada
- O relatório indicou nível seguro para uso, sem riscos diretos de invasão, injeção ou execução indevida de código

Esse resultado é esperado, considerando que o projeto opera apenas no front-end, sem backend.

Figura 6 - Resultado da análise de segurança no SonarCloud

3 – Plano de Refatoração

Com base nas issues identificadas pelas ferramentas de análise estática (ESLint e SonarCloud), foi elaborado um plano de refatoração focado em melhorar a manutenibilidade, legibilidade e boas práticas do código, sem alterar o funcionamento da aplicação.

3.1. Objetivos da Refatoração

O objetivo principal da refatoração foi aprimorar a qualidade, legibilidade e manutenção do código, seguindo as recomendações do SonarCloud e do ESLint. Especificamente, buscou-se:

- Reduzir a dívida técnica apontada pelas ferramentas de análise;
- Eliminar *code smells* e práticas desnecessárias identificadas;
- Tornar o código mais claro, organizado e fácil de manter;
- Aplicar padrões modernos de escrita em JavaScript;
- Facilitar futuras manutenções ou expansões da aplicação.

3.2. Principais Melhorias Planejadas

As melhorias planejadas foram implementadas de acordo com as issues identificadas:

- Remoção de variáveis e parâmetros não utilizados.
- Redução de múltiplas chamadas classList.add.
- Substituição de .forEach por for…of.

4 – Código Refatorado

Com base no plano de refatoração definido anteriormente, foram realizadas alterações no código-fonte com foco em melhorar a manutenibilidade, aumentar a legibilidade e eliminar alertas identificados pelo ESLint e SonarCloud. Essas mudanças incluíram a remoção de variáveis e parâmetros não utilizados, a consolidação de chamadas repetitivas de classList.add e a substituição de .forEach por estruturas for…of para percorrer arrays de forma mais clara e eficiente.

4.1. Correções Realizadas

- Remoção do parâmetro “e” não utilizado no evento “transitionend”, conforme apontado pelo ESLint.

Antes:

```
  todo.addEventListener("transitionend", (e) => {
    todo.remove();
  });
}
```

Depois:

```
  todo.addEventListener("transitionend", () => {
    todo.remove();
  });
}
```

- Redução de múltiplas chamadas classList.add.

Antes:

```
todo.classList.add("fall");
todo.classList.add("completed");
```

Depois:

```
todo.classList.add("fall", "completed");
```

- Substituição de .forEach por for…of.

Antes:

```
todos.forEach(function (todo) {
  // código
});
```

Depois:

```
for (const todo of todos) {
  // código
}
```

5 – Disponibilidade do Código Refatorado

- Com isso, o código foi refatorado podendo ver no repositório:

  Repositório com o código refatorado