Desenvolvimento Orientado a Testes (TDD) em Interfaces React: Um Estudo de Caso na Construção de uma Estante Virtual

Igor Felipe Monteiro de Souza, Frank Willian Cardoso de Oliveira, Marcelo Figueiredo Terenciani

[1]Instituto Federal do Paraná (IFPR) – Campus Paranavaí

20230006343@estudantes.ifpr.edu.br
{frank.willian, marcelo.terenciani}@ifpr.edu.br

Abstract. *The development of user interfaces requires solutions that support component structuring and testability. This study investigates how Test-Driven Development (TDD) relates to the internal structure of React components. To this end, a literary management system developed with TDD is compared with a legacy project not guided by testing practices. The research adopted a comparative case study, applying the Red-Green-Refactor cycle with Vitest and React Testing Library. The results indicate differences in the distribution of responsibilities between the analyzed systems. It is concluded that TDD was a practice associated with the modular organization observed in Booklog, although the identified differences cannot be attributed exclusively to its adoption.*

Resumo. *O desenvolvimento de interfaces demanda soluções que favoreçam a estruturação e a testabilidade dos componentes. Este trabalho investiga como o Test-Driven Development (TDD) se relaciona com a estrutura interna de componentes React. Para isso, compara-se um sistema de gestão literária desenvolvido com TDD a um projeto legado não orientado por testes. A pesquisa adotou um estudo de caso comparativo, aplicando o ciclo Red-Green-Refactor com Vitest e React Testing Library. Os resultados indicam diferenças na distribuição de responsabilidades entre os sistemas analisados. Conclui-se que o TDD constituiu uma prática associada à organização modular observada no Booklog, sem que seja possível atribuir exclusivamente a ele as diferenças identificadas.*

1. Introdução

O desenvolvimento de *software* tem exigido a construção de interfaces cada vez mais complexas, interativas e dinâmicas. Com a evolução do ecossistema *web*, bibliotecas como o React [Meta 2026] consolidaram-se como alternativas amplamente utilizadas no desenvolvimento de aplicações web. Contudo, à medida que a complexidade do código aumenta, surge o desafio de promover a confiabilidade e a manutenibilidade dessas aplicações em um ambiente de entregas contínuas [Banks and Porcello 2020].

Para mitigar tais riscos e favorecer a robustez dos sistemas, a prática de Desenvolvimento Orientado a Testes (*Test-Driven Development* – TDD), sistematizada e difundida por Kent Beck [Beck 2002], inverte o fluxo tradicional de codificação ao propor que os testes sejam escritos antes da implementação da funcionalidade. Princípios clássicos de engenharia de *software* sustentam que a camada de apresentação gráfica deve permanecer

desacoplada das regras de negócio para viabilizar a testabilidade. Ao conduzir o processo, o TDD pode favorecer o isolamento de funções, a aderência ao Princípio da Responsabilidade Única (*Single Responsibility Principle* – SRP) e a estruturação de um código modular e flexível [Martin 2011].

Embora o TDD seja uma prática reconhecida na Engenharia de Software, sua adoção no desenvolvimento de interfaces apresenta desafios técnicos e metodológicos. Entre eles, destacam-se a curva de aprendizado das ferramentas de teste, a adaptação do processo de desenvolvimento e a dificuldade de simular interações e comportamentos visuais isoladamente [Irvine 2022]. Nesse contexto, a escrita de testes somente após a implementação pode exigir refatorações posteriores e dificultar a verificação de componentes que concentram renderização, gerenciamento de estado e comunicação com serviços externos. Essa concentração de responsabilidades pode comprometer a testabilidade e a manutenção do código [Martin 2011, Irvine 2022].

Entre os trabalhos examinados nesta pesquisa, identifica-se uma oportunidade de investigação sobre a relação entre a aplicação do ciclo *Red-Green-Refactor* e a organização estrutural de componentes React. Embora as pesquisas selecionadas abordem estratégias de testes automatizados, cobertura de código e verificação de interfaces, a análise da distribuição de responsabilidades durante o desenvolvimento orientado por testes não constitui seu foco principal. Nesse sentido, torna-se pertinente examinar artefatos de software produzidos com e sem a adoção do TDD, buscando identificar diferenças relacionadas à modularidade, ao acoplamento e à separação entre a apresentação visual e as regras de negócio.

Diante desse contexto, este trabalho tem como objetivo desenvolver o *front-end* de um sistema web de gestão literária, denominado *Booklog*, utilizando o TDD, e comparar sua estrutura à de um projeto legado não orientado por testes, denominado *Web Auction*. A investigação busca identificar diferenças na organização de componentes React, considerando indicadores como linhas de código, dependências, gerenciamento de estado e distribuição de responsabilidades. Como contribuição, o estudo apresenta uma análise comparativa de artefatos de software e documenta a aplicação do ciclo *Red-Green-Refactor* com *Vitest* e *React Testing Library*, oferecendo subsídios para discutir a relação entre testabilidade e organização modular de interfaces.

2. Desenvolvimento Orientado a Testes (*Test-Driven Development* – TDD)

O TDD é uma técnica de desenvolvimento de software associada às práticas ágeis, caracterizada pela inversão do fluxo tradicional de codificação, na qual os testes são escritos antes da implementação do código. Sistematizada e difundida por Kent Beck [Beck 2002] como uma das práticas centrais da Programação Extrema (*Extreme Programming* – XP), essa abordagem não se limita à identificação de falhas. Em vez de apenas validar se uma lógica preexistente funciona, o TDD também pode apoiar decisões de projeto, incentivando a definição prévia dos comportamentos esperados e das condições necessárias à sua verificação [Irvine 2022].

O funcionamento dessa técnica organiza-se em um ciclo iterativo, conhecido como *Red-Green-Refactor*. Na etapa vermelha (*Red*), o desenvolvedor escreve um teste para um comportamento que ainda não foi implementado, verificando sua falha. Na sequência, durante a etapa verde (*Green*), produz-se a menor quantidade de código necessária para que o teste seja aprovado. Por fim, na etapa de refatoração (*Refactor*), o código pode ser reorganizado para melhorar sua clareza e estrutura, enquanto a execução dos testes auxilia na identificação de possíveis regressões nos comportamentos verificados [Beck 2002].

No contexto do objeto estudado, o TDD constitui a prática de desenvolvimento adotada na implementação do *Booklog*. A exigência de testabilidade prévia é utilizada não apenas para verificar a consistência das regras de negócio, mas também como referência para examinar a organização das responsabilidades entre os componentes da aplicação. Ao incentivar a validação antecipada, o ciclo *Red-Green-Refactor* pode contribuir para decisões relacionadas à separação entre lógica de negócio, gerenciamento de estado e renderização visual [Martin 2011].

3. O Ecossistema React e Componentização

A aplicação do TDD no desenvolvimento de interfaces pode ser examinada em ambientes que utilizam bibliotecas baseadas em componentes, como o React. Consolidado entre as principais tecnologias da indústria de *software* [Stack Overflow 2024], o React oferece recursos para a construção de interfaces declarativas e para o gerenciamento de estados e atualizações. A biblioteca adota um modelo baseado em componentes, permitindo organizar a apresentação visual em unidades reutilizáveis [Meta 2026].

As características do React incluem o modelo declarativo de construção de interfaces e os mecanismos de gerenciamento de estado. Ao adotar esse modelo, o React permite descrever a interface a partir dos dados e estados da aplicação, realizando as atualizações no Modelo de Objetos de Documentos (*Document Object Model* – DOM). Além disso, a organização em componentes possibilita delimitar responsabilidades e compartilhar informações por meio de propriedades, enquanto os estados podem ser mantidos localmente ou compartilhados entre diferentes partes da aplicação [Banks and Porcello 2020].

No contexto dos testes automatizados, a organização em componentes possibilita a utilização de ferramentas como o Vitest [Vitest Contributors 2026], responsável pela execução dos testes, e a React Testing Library [Dodds 2026], que permite renderizar componentes e verificar comportamentos observáveis por meio de interações simuladas. Assim, essas ferramentas podem apoiar a aplicação do ciclo *Red-Green-Refactor*, permitindo definir comportamentos esperados antes da implementação e verificar respostas dos componentes durante o desenvolvimento.

4. Trabalhos Relacionados

Para mapear as abordagens adotadas na literatura, realizou-se uma revisão exploratória da literatura focada no uso de testes no desenvolvimento de interfaces *web*. Este levantamento incorporou artigos recentes e trabalhos de conclusão de curso. Essa escolha metodológica justifica-se pela necessidade de examinar relatos práticos que documentam as particularidades de bibliotecas específicas de *front-end*. O levantamento não seguiu um protocolo sistemático de busca e seleção, não abrangendo necessariamente toda a produção científica sobre o tema.

Uma primeira vertente de estudos recentes investiga a otimização da cobertura de testes e sua integração ao fluxo de desenvolvimento por meio de Inteligência Artificial (IA). O trabalho de [Mumtaz et al. 2025], aplicado ao domínio de transporte público, investigou a integração do *framework* React a práticas ágeis utilizando ferramentas de IA Generativa para apoiar a elaboração de casos de teste. Sob uma perspectiva similar de automação, [Jiang 2025] buscou atenuar a complexidade de manutenção de suítes de teste em arquiteturas Vue.js mediante um algoritmo de aprendizado por reforço (Sarsa), priorizando a eficiência na geração de cenários de verificação.

Uma segunda vertente aborda testes automatizados e desempenho em arquiteturas baseadas em componentes. [Moroz 2019] investigou testes unitários em aplicações React e Redux, utilizando a biblioteca Enzyme em fluxos de Integração Contínua e Entrega Contínua (*Continuous Integration/Continuous Delivery* – CI/CD). [Nayim et al. 2023] avaliaram a escalabilidade de arquiteturas monolíticas e de microsserviços por meio de testes de estresse com aplicações com React. Adicionalmente, [Kurkinen 2018] documentou a transição de verificações manuais para testes automatizados com Ember.js, apontando ganhos na detecção de regressões em um sistema corporativo.

Ao comparar os trabalhos, observa-se convergência na adoção de arquiteturas de componentes, mas distinções quanto ao momento de aplicação dos testes e a sua finalidade. Para sistematizar essa análise na Tabela 1, os trabalhos foram classificados segundo quatro critérios: (i) *Ecossistema*, que identifica a tecnologia principal de interface; (ii) *Abordagem de Testes*, que distingue se os testes foram criados após o código (*Test-Last*), gerados por modelos computacionais ou escritos previamente (*Test-First*/TDD); (iii) *Foco da Verificação*, que explicita o alvo principal dos testes no estudo; e (iv) *Análise no Design de Componentes*, que indica se foi examinado a estrutura interna da interface.

Tabela 1. Síntese comparativa entre os trabalhos relacionados e este estudo.

Estudo	**Ecossistema**	**Abordagem / Foco**	**Foco da Verificação**	***Design***
[Kurkinen 2018]	Ember.js	Pós-código (*Test-Last*)	Automação de QA e prevenção de regressões	Não
[Moroz 2019]	React e Redux (Enzyme)	Pós-código (*Test-Last*)	Estado interno e métodos em pipeline CI/CD	Não
[Nayim et al. 2023]	React	Pós-código (*Test-Last*)	Desempenho e estresse	Não
[Jiang 2025]	Vue.js	Geração por IA (Sarsa)	Otimização algorítmica de cobertura	Não
[Mumtaz et al. 2025]	React	Geração Assistida por IA	Criação de casos de teste em fluxo ágil	Não
Este estudo	**React**	**Pré-código (TDD)**	**Comportamento observável pelo usuário**	**Sim**

Conforme sintetizado na Tabela 1, a principal lacuna observada nas referências analisadas reside no tratamento dos testes prioritariamente como um mecanismo de verificação posterior. Além disso, abordagens voltadas ao React que se apoiam em ferramentas da geração anterior, como o estudo de [Moroz 2019] com a biblioteca Enzyme, podem apresentar maior dependência de detalhes internos de implementação (como a estrutura de estados privados), diferindo das práticas atuais voltadas ao comportamento percebido pelo usuário. Consequentemente, identificou-se, entre os trabalhos examinados, espaço para investigar o TDD no *front-end* como um mecanismo de concepção arquitetural.

Este trabalho busca contribuir para essa discussão ao adotar o ciclo *Red-Green-Refactor* desde a etapa inicial de construção da interface. Ao combinar o React à *React Testing Library*, a abordagem prioriza a verificação do comportamento visível ao usuário em detrimento de detalhes internos de implementação. Mediante o desenvolvimento do sistema *Booklog* e seu contraste estrutural com um projeto legado desprovido de testes (*Web Auction*), a investigação analisa qualitativamente as diferenças observadas no acoplamento e na distribuição de responsabilidades entre os componentes dos dois sistemas.

5. Metodologia

Esta pesquisa caracteriza-se como um estudo de caso comparativo de natureza qualitativa e exploratória [Wohlin et al. 2012]. Segundo [Wazlawick 2014], abordagens descritivas e estudos de caso em computação permitem investigar e caracterizar fenômenos contemporâneos em profundidade dentro de seu contexto real de aplicação. Nesse cenário, o desenvolvimento dos projetos atua como o ambiente prático de observação, possibilitando analisar qualitativamente as diferenças na estruturação e no acoplamento de componentes React desenvolvidos com e sem a adoção do Desenvolvimento Orientado a Testes.

Para alcançar o objetivo, o processo de engenharia de *software* seguiu um modelo baseado no gerenciamento ágil com práticas do XP [Beck 2002] e do *Scrum* [Schwaber and Sutherland 2020]. As etapas do projeto foram organizadas em ciclos curtos de trabalho (*Sprints*), o levantamento de requisitos por meio de Histórias de Usuário, a modelagem arquitetural utilizando diagramas da Linguagem de Modelagem Unificada (*Unified Modeling Language* – UML) e a implementação de código guiada por testes.

5.1. Preparação

O objeto de estudo é o *front-end* de uma aplicação *web* denominada *Booklog*, um sistema de gestão de acervo literário no qual os usuários podem adicionar obras à sua estante e acompanhar o progresso de leitura. A construção dessa interface serve como base empírica para investigar qualitativamente a organização arquitetural de componentes desenvolvidos com TDD. Adicionalmente, o delineamento incorpora uma análise comparativa com um projeto legado (denominado *Web Auction*), desenvolvido previamente pelo mesmo autor sem a adoção de testes automatizados, atuando como contraponto prático para a discussão dos resultados observados.

A seleção das tecnologias baseou-se na busca por viabilizar a escalabilidade e o suporte nativo à automação em ecossistemas de componentes reativos. O desenvolvimento da interface utilizou a biblioteca React [Meta 2026], com Vite [Vite Contributors 2026]. Para executar os testes, optou-se pelo uso conjunto do Vitest [Vitest Contributors 2026] e do *React Testing Library* [Dodds 2026], escolhida por sua capacidade de renderizar e simular interações sob a ótica do usuário final.

Os critérios e métricas utilizados para avaliar a abordagem adotada dividem-se em indicadores quantitativos e qualitativos. No âmbito quantitativo, empregou-se a métrica de cobertura de código fornecida pelo Vitest para mensurar a abrangência das linhas e ramificações exercitadas pela suíte de testes. Qualitativamente, a análise fundamentou-se nos princípios de código limpo e arquitetura de *software* [Martin 2011], tendo como critérios o nível de desacoplamento alcançado entre regras de negócio e apresentação gráfica, a aderência ao SRP e a legibilidade dos componentes em contraste com o sistema legado.

5.2. Execução

O ambiente de execução fundamentou-se em uma arquitetura no lado do cliente (*client-side*), estruturada como uma Aplicação de Página Única (*Single Page Application* – SPA). Para delimitar o escopo, optou-se por prescindir de serviços remotos de *back-end* ou bancos de dados externos. A persistência foi implementada por meio do recurso nativo *localStorage*, utilizando JSON. O roteamento foi gerenciado pelo React Router e o versionamento do código-fonte foi mantido com Git e GitHub[1]. A implementação foi orientada pelo ciclo *Red-Green-Refactor*. Para cada componente do *Booklog*, formulava-se inicialmente um teste para o comportamento esperado (etapa *Red*).

Após a confirmação da falha, implementava-se a quantidade mínima de código necessária para a aprovação do teste (etapa *Green*). Em seguida, o código passava por refatorações (etapa *Refactor*), visando ao aprimoramento da clareza e da manutenibilidade, com nova execução dos testes para verificar possíveis regressões. A coleta de dados ocorreu ao longo dos ciclos de desenvolvimento (*Sprints*), com registro dos artefatos produzidos. Foram catalogados arquivos de teste, relatórios técnicos de execução dos testes (*test runners*) e estruturas finais dos componentes React. Em paralelo, foram selecionados artefatos do *Web Auction* para comparação estrutural com os componentes do *Booklog*.

5.3. Análise dos dados

Os dados obtidos durante a execução passaram por um tratamento qualitativo focado na separação de responsabilidades. Antes da análise comparativa, o código-fonte do *Booklog* foi mapeado e categorizado, distinguindo as regras de negócio e a manipulação de estado da renderização visual. Esse procedimento preparou os artefatos desenvolvidos com TDD para comparação com o projeto legado. Também foram considerados os relatórios de cobertura de testes, utilizados para descrever os trechos de código executados pela suíte. Esses indicadores complementaram a análise estrutural, sem serem tratados como medidas diretas de modularidade, acoplamento ou ausência de defeitos.

O método de avaliação baseou-se na análise descritiva dos componentes e na cobertura de código obtida pelos testes, sem considerá-la indicador de qualidade arquitetural ou ausência de falhas. A estrutura do *Booklog* foi comparada qualitativamente ao sistema legado em duas frentes de escala: componentes de escopo reduzido e módulos de maior complexidade. O objetivo desse delineamento foi investigar se as diferenças estruturais são mais evidentes nos módulos complexos, considerando o acoplamento e a presença de concentração de responsabilidades em componentes (*God Components*).

A análise comparativa entre o *Booklog* e o sistema legado *Web Auction* foi organizada em duas frentes: (i) componentes menores, comparando os cartões de itens (*BookCard.jsx* e *AuctionCard.js*); e (ii) módulos mais complexos, voltados à orquestração de telas e regras de negócio. Nesta segunda frente, apesar dos diferentes domínios, comparou-se a abordagem monolítica do sistema legado (*AuctionDetail.js*) com a arquitetura distribuída do *Booklog* (*ShelfView.jsx*, *useShelf.js* e *StatusModal.jsx*). Em ambas, foram considerados quatro indicadores para caracterizar a modularidade e o acoplamento: linhas de código, dependências externas (*imports*), *hooks* gerenciados na camada visual e responsabilidades por arquivo.

[1] https://github.com/TCC-2026-Igor-Felipe/booklog

6. Resultados

O desenvolvimento do sistema *Booklog* orientado por testes resultou em uma arquitetura modular executada integralmente no lado do cliente. Esta seção apresenta os dados empíricos observados na implementação dos módulos, os índices de cobertura alcançados e o contraste métrico direto entre artefatos do *Booklog* e do sistema legado *Web Auction*.

6.1. Qualidade Estrutural e Cobertura de Código

O primeiro resultado refere-se à organização da lógica de manipulação de dados, gerenciada pelo isolamento de regras de negócio em *hooks* customizados, como o *useShelf*. Para ilustrar a aplicação prática da etapa *Red* na arquitetura, o Código 1 apresenta um caso de teste extraído da suíte de validação do projeto.

```
it('deve adicionar um livro à estante com status de leitura', () => {
    const { result } = renderHook(() => useShelf());
    const novoLivro = { titulo: 'O Hobbit', autor: 'J.R.R. Tolkien' };

    act(() => {
        result.current.adicionarLivro(novoLivro, { statusLeitura: 'QUERO_LER' });
    });

    expect(result.current.estante).toHaveLength(1);
    expect(result.current.estante[0].titulo).toBe('O Hobbit');
    expect(result.current.estante[0].statusLeitura).toBe('QUERO_LER');

    const estanteSalva = JSON.parse(window.localStorage.getItem('booklog_estante'));
    expect(estanteSalva).toHaveLength(1);
    expect(estanteSalva[0].statusLeitura).toBe('QUERO_LER');
});
```

Código 1. Exemplo de caso de teste automatizado para o *hook useShelf*.

O Código 1 define o comportamento esperado da função de adição de livros, validando a atualização do estado reativo e a persistência na API de armazenamento local do navegador antes de a lógica ser codificada no componente. A elaboração sistemática de testes contemplou critérios estabelecidos, como o tratamento contra obras duplicadas e o controle isolado de itens favoritos. A Tabela 2 consolida o relatório de cobertura gerado pela ferramenta Vitest após a conclusão dessa etapa.

Tabela 2. Relatório de cobertura de testes gerado pelo *framework* Vitest.

Camada / Módulo (*File / Group*)	**Declarações (*% Stmts*)**	**Ramificações (*% Branch*)**	**Funções (*% Funcs*)**	**Linhas (*% Lines*)**
Componentes Visuais (6 arquivos)	100,00	100,00	100,00	100,00
***Hooks* Customizados**				
useSearch.js	100,00	100,00	100,00	100,00
useCustomLists.js	100,00	96,66	100,00	100,00
useShelf.js	100,00	92,59	100,00	100,00
Telas (*Views*)				
Demais telas (5 arquivos)	100,00	100,00	100,00	100,00
ProfileView.jsx	100,00	97,29	100,00	100,00
Total Geral (*All files*)	**100,00**	**98,55**	**100,00**	**100,00**

Fonte: Elaborado pelos autores (2026).

Conforme observado na Tabela 2, o desenvolvimento alcançou 100% de cobertura em linhas de código, declarações e funções. Quanto à cobertura de ramificações (*branches*), obteve-se o índice de 98,55%, com variações pontuais restritas aos *hooks* *useCustomLists.js* (96,66%) e *useShelf.js* (92,59%) e à tela *ProfileView.jsx* (97,29%), decorrentes de estruturas de controle defensivas e valores padrão que dispensaram fluxos adicionais. Ressalta-se, contudo, que a cobertura de testes reflete apenas a quantidade de código executada, não servindo como indicador de qualidade arquitetural, baixo acoplamento ou ausência de defeitos. No *Booklog*, esses números indicam a abrangência da verificação, enquanto a modularidade observada nos componentes foi estimulada pelo ciclo do TDD.

6.2. Consistência da Interface de Usuário

Outro resultado refere-se à organização da camada visual da aplicação, indicando que a aplicação do TDD contemplou tanto a manipulação de estado quanto a estruturação dos componentes de apresentação. A interface construída reflete o consumo dinâmico dos estados previamente testados e uma organização desacoplada entre renderização e regras de negócio. A Figura 1 apresenta a interface principal da estante pessoal (*ShelfView*).

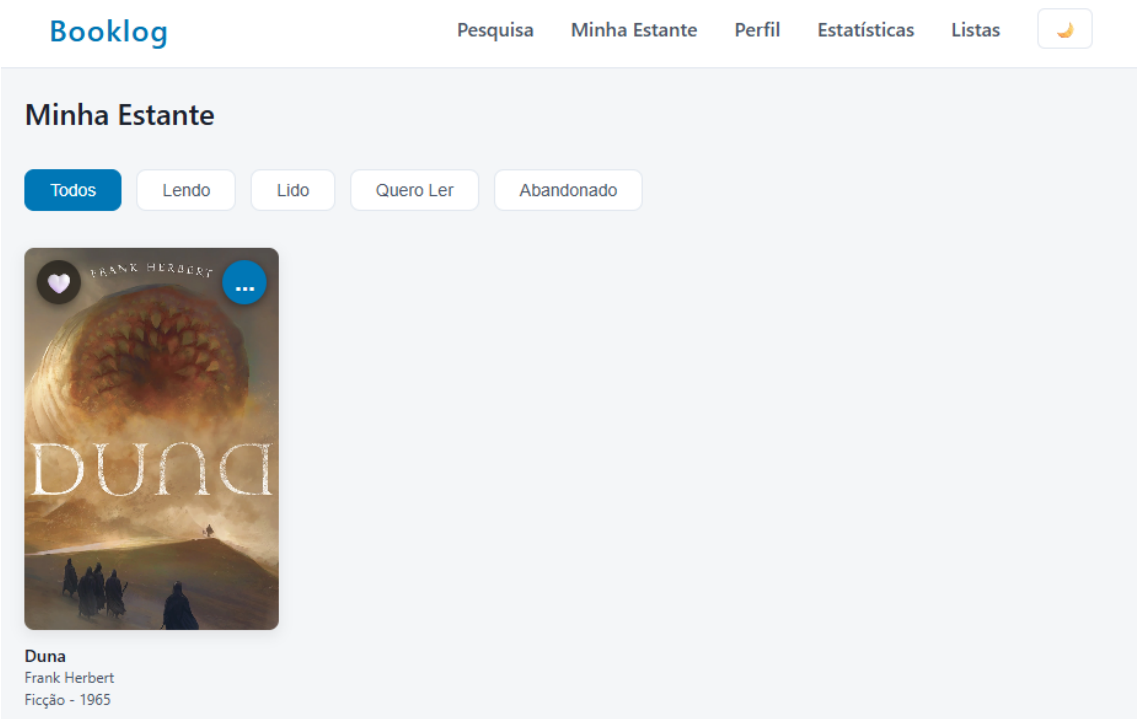


Figura 1. Interface "Minha Estante" renderizando as obras adicionadas.
Fonte: Elaborado pelos autores (2026).

A interface evidencia a separação entre componentes contêiner e de apresentação. O componente de exibição do livro (*BookCard*) atua na renderização visual a partir das propriedades recebidas, delegando ao componente pai a interação com o estado global e o armazenamento local. Essa estrutura permitiu que elementos secundários, como os modais de atualização de leitura (*StatusModal*), operassem de forma isolada, caracterizando a separação de responsabilidades observada.

6.3. Análise Comparativa de Artefatos de Software

Para investigar as diferenças estruturais entre sistemas desenvolvidos com e sem testes prévios, confrontou-se o código-fonte do *Booklog* com o projeto legado *Web Auction* nas duas frentes de escala definidas metodologicamente. A Tabela 3 consolida as métricas observadas nos pares e grupos de artefatos analisados.

Tabela 3. Comparação de métricas entre módulos do *Web Auction* e do *Booklog*.

Critério / Indicador	**Frente 1: Escopo Reduzido (Cartões)**		**Frente 2: Escopo Abrangente**	
	Web Auction **AuctionCard.js**	*Booklog* **BookCard.jsx**	*Web Auction* **AuctionDetail.js**	*Booklog* **Tríade Modular**
Abordagem	Sem testes	TDD	Sem testes	TDD
Linhas de Código	151 linhas	65 linhas	713 linhas	261 linhas totais: 53 na *ShelfView*, 79 no *useShelf* e 129 no *StatusModal*
Dependências (*Imports*)	6 importações	5 importações	19 importações	4 na *View*, 1 no *Hook* e 3 no *Modal*
Chamadas de Hooks	N/A (Componente de apresentação)	N/A (Componente de apresentação)	16 chamadas (10 *useState*, 2 *useEffect*, 1 *useRef*, 3 externos)	2 na *View*, 0 no *Hook* e 3 no *Modal*
Distribuição de Responsabilidades	1 (Apresentação visual)	1 (Apresentação visual)	5 (Renderização UI, modal de lances, chamadas API/WS, regras e DOM)	1 por módulo (UI na *View*; regras/-persistência no *Hook*; diálogo no *Modal*)
Padrão Arquitetural	Baixo acoplamento (delegação externa)	Baixo acoplamento (delegação externa)	Monolítico (*God Component*)	Decomposição em módulos

Na primeira frente, os cartões *AuctionCard.js* (151 linhas e 6 importações) e *BookCard.jsx* (65 linhas e 5 importações) apresentam organização semelhante quanto à distribuição de responsabilidades, embora difiram em extensão. Ambos concentram a apresentação visual e delegam outras operações a componentes externos. Na segunda frente, as diferenças são mais evidentes: o componente legado *AuctionDetail.js* reúne 713 linhas, 19 dependências externas e 16 chamadas a *hooks*, concentrando renderização, controle de diálogos, manipulação do Modelo de Objetos do Documento (*Document Object Model* – DOM), conexões de rede e validação de lances.

No *Booklog*, por sua vez, o gerenciamento da tela e as regras de negócio associadas foram organizados em três arquivos: *ShelfView.jsx*, responsável pela visualização (53 linhas, 4 importações e 2 *hooks*); *useShelf.js*, dedicado à lógica e à persistência (79 linhas e 1 importação); e *StatusModal.jsx*, voltado ao diálogo modal (129 linhas e 3 importações). Juntos, esses arquivos totalizam 261 linhas, com responsabilidades distribuídas entre apresentação, gerenciamento de estado e interação modal. Essa organização contrasta com a concentração de funções em um único componente observada no sistema legado, sem permitir atribuir tal diferença exclusivamente à adoção do TDD.

7. Discussão

A análise comparativa consolidada na Tabela 3 indica que diferenças estruturais observadas variam de acordo com a escala e a densidade da implementação. Na primeira frente, voltada a componentes de escopo reduzido e puramente visuais (*AuctionCard.js* e *BookCard.jsx*), a diferença arquitetural entre os sistemas mostra-se sutil, visto que a própria natureza declarativa do React já favorece um encapsulamento básico de apresentação. Nesses casos, a adoção do TDD esteve associada principalmente à organização do fluxo de desenvolvimento. A formulação prévia de testes orientou o estabelecimento de fronteiras lógicas, favorecendo o componente à delegação de eventos e ao emprego controlado de dublês de teste (*test doubles*), em consonância com a proposta do TDD.

Na segunda frente de escala, contudo, identificaram-se diferenças estruturais mais evidentes entre os módulos analisados. No sistema legado *Web Auction*, o componente *AuctionDetail.js* apresentou concentração de responsabilidades, caracterizando um componente sobrecarregado (*God Component*). A concentração de chamadas de rede, manipulação direta do DOM, diálogo modal de lances e validações de domínio em um único módulo de 713 linhas pode dificultar a elaboração de testes unitários devido à concentração de responsabilidades. Em contrapartida, no *Booklog*, a possibilidade de testar as regras de negócio sem renderizar toda a interface estimulou a decomposição dessa estrutura entre visualização (*ShelfView.jsx*), controle de estado e persistência (*useShelf.js*) e interação modal (*StatusModal.jsx*).

Essas constatações complementam a literatura analisada e diferenciam-se das abordagens voltadas à automação retrospectiva. Enquanto investigações ([Mumtaz et al. 2025, Jiang 2025] exploram o emprego de Inteligência Artificial prioritariamente na síntese posterior de testes para ganhos de produtividade, este estudo concentrou-se na aplicação do TDD em sua dinâmica disciplinar. Diferentemente da geração automatizada sobre códigos já prontos, que pode atingir alta cobertura mesmo em módulos monolíticos, a escrita prévia do teste estimula um esforço analítico que atua preventivamente sobre a arquitetura dos componentes antes da codificação.

Por fim, cabe explicitar as variáveis de confusão inerentes a este estudo. Como o *Booklog* e o *Web Auction* são sistemas de domínios diferentes e desenvolvidos em momentos distintos, as diferenças arquiteturais observadas não podem ser atribuídas exclusivamente ao TDD, devendo-se reconhecer o amadurecimento técnico do próprio desenvolvedor entre um projeto e outro. Todavia, a experiência prática indica que o ciclo *Red-Green-Refactor* operou como o mecanismo que auxiliou essa evolução técnica, sugerindo que o investimento inicial na escrita prévia de testes favorece a organização modular do código e contribui para mitigar o acúmulo de dívida técnica nos cenários avaliados.

8. Conclusão

Este trabalho teve como objetivo desenvolver o *front-end* de um sistema de gestão literária, denominado *Booklog*, orientado pelo TDD, comparando-o qualitativamente ao sistema legado *Web Auction* para analisar as diferenças na estruturação de componentes React. Os resultados indicaram que a disciplina do ciclo *Red-Green-Refactor* estimulou uma postura preventiva no planejamento do código, cujos efeitos práticos variaram conforme a escala do artefato: mostraram-se sutis em componentes visuais de escopo reduzido, mas apresentaram relevância arquitetural em módulos mais densos.

A principal contribuição observada no desenvolvimento do *Booklog* residiu na distribuição de responsabilidades entre componentes, em contraste com a concentração de funções identificada no módulo *AuctionDetail.js* do sistema legado *Web Auction*. No *Booklog*, desenvolvido sob a abordagem TDD, as responsabilidades foram distribuídas entre visualização (*ShelfView.jsx*), interação modal (*StatusModal.jsx*) e gerenciamento de estado e persistência (*useShelf.js*), resultando em módulos individualmente verificáveis.

Quanto às limitações metodológicas, destaca-se que a análise comparativa envolveu sistemas de domínios distintos, desenvolvidos em momentos diferentes por um único pesquisador. Dessa forma, as diferenças observadas no *Booklog* não podem ser atribuídas exclusivamente ao TDD, podendo estar associadas também ao amadurecimento técnico do desenvolvedor no intervalo entre os projetos. Ademais, a aplicação operou exclusivamente com persistência local no navegador, sem serviços remotos acoplados, o que delimitou a análise a um ambiente local e restringiu conclusões sobre cenários complexos com APIs de *back-end*.

Para investigar as limitações identificadas, sugere-se a condução de experimentos controlados envolvendo múltiplos desenvolvedores submetidos aos mesmos requisitos de domínio, de modo a controlar a variável da curva de aprendizagem individual e mensurar indicadores quantitativos de acoplamento, esforço e tempo de desenvolvimento. Recomenda-se, por fim, investigar a aplicação do TDD em sistemas de maior escala integrados a arquiteturas de microsserviços e APIs assíncronas, bem como avaliar empiricamente o papel das suítes de testes automatizados como documentação viva na manutenção evolutiva de interfaces.

Declaração de uso de ferramentas de Inteligência Artificial

Os autores declaram que utilizaram ferramentas de Inteligência Artificial (IA), especificamente o Gemini 3.1 Pro e o ChatGPT (GPT-6), como apoio na elaboração deste manuscrito. O Gemini foi utilizado na síntese e reestruturação do Trabalho de Conclusão de Curso (TCC) para este artigo, principalmente na revisão textual e formatação em LaTeX. O ChatGPT foi empregado na revisão da linguagem acadêmica, concisão textual e adequação das afirmações aos limites metodológicos da pesquisa. A utilização dessas ferramentas restringiu-se ao suporte à produção acadêmica, permanecendo sob responsabilidade dos autores a análise, a validação e o conteúdo final do trabalho, bem como sua originalidade, precisão e integridade ética.

Referências

Banks, A. and Porcello, E. (2020). *Learning React: Modern Patterns for Developing React Apps*. O'Reilly Media, Sebastopol, 2nd edition.

Beck, K. (2002). *Test-Driven Development: By Example*. Addison-Wesley Professional, Boston.

Dodds, K. C. (2026). React Testing Library. Disponível em: https://testing-library.com. Acesso em: 30 set. 2026.

Irvine, D. (2022). *Mastering React Test-Driven Development*. Packt Publishing, Birmingham.

Jiang, B. (2025). The Application of Vue.js Framework Technology in Multidomain Automated Testing Systems. *Automatic Control and Computer Sciences*, 59(4):455–466.

Kurkinen, T. (2018). Testing Automation to Increase Product Quality. Bachelor's thesis, Metropolia University of Applied Sciences, Helsinki.

Martin, R. C. (2011). *Código Limpo: Habilidades Práticas do Agile Software*. Alta Books, Rio de Janeiro.

Meta (2026). React: The Library for Web and Native User Interfaces. Disponível em: https://react.dev. Acesso em: 30 set. 2026.

Moroz, B. (2019). Unit Test Automation of a React-Redux Application with Jest and Enzyme. Bachelor's thesis, South-Eastern Finland University of Applied Sciences, Mikkeli.

Mumtaz, A., Haider, A., Haroon, M., and Ahmed, A. (2025). Application Development with Agile and Django-React Framework: A Case Study on Automated Reliability Testing with Bus Ticketing System. *Bulletin of the Tomsk Polytechnic University. Industrial Cybernetics*, 3(3):44–50.

Nayim, N. N., Karmakar, A., Ahmed, M. R., Saifuddin, M., Kabir, M. H., and Hasan, M. N. (2023). Performance Evaluation of Monolithic and Microservice Architecture for an E-commerce Startup. In *Proceedings of the 26th International Conference on Computer and Information Technology (ICCIT)*, pages 1–6. IEEE.

Schwaber, K. and Sutherland, J. (2020). O Guia do Scrum: O Guia Definitivo para o Scrum – As Regras do Jogo. Disponível em: https://scrumguides.org. Acesso em: 30 set. 2026.

Stack Overflow (2024). 2024 Stack Overflow Developer Survey. Disponível em: https://survey.stackoverflow.co/2024. Acesso em: 30 set. 2026.

Vite Contributors (2026). Vite: Next Generation Frontend Tooling. Disponível em: https://vite.dev. Acesso em: 30 set. 2026.

Vitest Contributors (2026). Vitest: A Blazing Fast Unit Test Framework Powered by Vite. Disponível em: https://vitest.dev. Acesso em: 30 set. 2026.

Wazlawick, R. S. (2014). *Metodologia de Pesquisa para Ciência da Computação*. Elsevier, Rio de Janeiro, 2nd edition.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer, Berlin.