修士学位論文

題目

フロントエンドフレームワークを採用したOSSプロジェクトにおけるテスト活用の現状と課題の分析

指導教員

楠本 真二 教授

報告者

久保 光生

令和7年1月28日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

令和 6 年度 修士学位論文

フロントエンドフレームワークを採用した OSS プロジェクトにおけるテスト活用の現状と課題の分析

久保 光生

内容梗概

近年複雑化する Web 開発において，コンポーネントベースの設計やデータバインディング，状態管理といった特徴を持つフロントエンドフレームワークは主要な地位を占める．フレームワークを用いたプロジェクトにおいては，テストは信頼性の高い動作を保証するために重要であり，特にコンポーネントのテストはフレームワーク公式のテストガイドにおいて重要視されている．しかしながら，フロントエンドフレームワークにおけるテストは複雑であり，特に E2E テストの採用率の低さや，コンポーネントテストの導入障壁といった課題がある．これらの課題の実態を明らかにするため，本研究では 4 つの RQ を設定し，フロントエンドフレームワークにおけるテストの現状を多角的に調査した．調査のため，GitHub からフレームワークを使用した OSS リポジトリを 444 件収集し，リポジトリ内にあるテストを実行した．テストの実行結果や他の指標を分析することで，テスト採用率，テストの品質，バグ報告数とテストの数やテストの品質との関係，テスト更新頻度などを調査した．調査の結果，テストの課題として，コンポーネントは他のファイルよりもテストされにくい傾向にあることや，フロントエンドフレームワークに特有のテスト手法をうまく扱えていないことなど，いくつかの課題が明らかになった．また，調査結果をもとに，テスト保守について開発者が知るべき事項や，保守の観点から推奨されるテスト手法など，テストの改善に向けた具体的なフィードバックを示した．

主な用語

フロントエンドフレームワーク, ソフトウェアテスト, オープンソースソフトウェア，リポジトリマイニング

目次

図目次

表目次

1 はじめに

近年，Web アプリケーションは複雑化しており，高度なユーザーインタフェース（UI）やインタラクティブな動作が求められるようになっている．このようなニーズに対応するため，近年 React，Vue.js，Angular，Svelte といったフロントエンドフレームワークが広く採用されている．これらのフレームワークは，コンポーネントベースの設計やデータバインディング，状態管理といった特徴を持つことで，開発効率を向上させるとともに，コードの再利用性や保守性を高めている．フレームワークを用いたプロジェクトにおいてテストは重要である．これは，フレームワークが提供するコンポーネントの再利用性や状態管理の仕組みが，意図しない副作用や予期しない挙動を引き起こす可能性があるためである．フロントエンドフレームワークにおいては，主に単体テスト，結合テスト，E2E テストが使用される．特に，単体テストまたは結合テストの一種であるコンポーネントテストは，フレームワーク公式のテストガイドにおいて重要視されている．一方で，フロントエンドフレームワークを用いた開発において，適切なテストの実施には依然として多くの課題が残されている．特に，コンポーネント単位でのテストは，従来のユニットテストや結合テストとは異なるアプローチが必要とされ，テストの導入や保守に高いコストがかかる傾向がある．また，その他にも，E2E（エンドツーエンド）テストの採用率の低さや，テストの保守性に関する課題が存在する．しかしながら，このような課題について調査した研究は今まで存在しなかった．また，フロントエンドフレームワークを用いた実際のプロジェクトにおいて，テストがどの程度採用されているのか，またその品質がどのような水準にあるのかは明らかになっていない．これらの課題や，テストの実態を明らかにするため，本研究では 4 つの RQ を設定し，フロントエンドフレームワークにおけるテストの現状を多角的に調査した．具体的には，GitHub からフレームワークを使用した OSS リポジトリを 444 件収集し，テストの実行結果や他の指標を分析した．

本研究では以下の 4 つの Research Questions を設定し，調査を行った．

- RQ1: Web フロントエンドフレームワークを採用したプロジェクトにおけるテストの使用率は？
- RQ2: Web フロントエンドフレームワークを採用したプロジェクトにおけるテストの品質は？
- RQ3: Web フロントエンドフレームワークを採用したプロジェクトにおいて，バグ報告数とテストケースの数・品質等との関係は？
- RQ4: Web フロントエンドフレームワークを採用したプロジェクトにおけるテストの更新頻度や，ソースコードの更新頻度との関係は？

RQ1 では，GitHub から収集したプロジェクトにおけるテストの採用率を分析した．その結果，単体・結合テストは広く採用されている一方で，コンポーネントテストや E2E テストの採用率が低いといった課題が明らかになった．

RQ2 では，GitHub から収集したプロジェクトでテストを実行し，テストの成功率やカバレッジを測定した．その結果，E2E テストの成功率が比較的低いことや，コンポーネント定義ファイルのカバレッジが 0 である，すなわちコンポーネントを全くテストできていないプロジェクトが多いという課題が明らかになった．また，テストの失敗要因の多くはフロントエンドフレームワークに特有の実行時エラーであり，これらを知ることはテストの保守にとって重要であることを示した．

RQ3 では，テストケースの数やカバレッジと，バグ報告数との相関を分析した．その結果，バグ報告数とテストの数には相関は見られなかった．一方で，カバレッジが高いプロジェクトほどバグ報告が多い傾向が確認され，カバレッジがバグの発見に寄与する可能性が示唆された．

RQ4 では，単体・結合テストをコンポーネントテストとその他のテストの 2 種類に分け，テストファイルの更新頻度や，テストファイルの更新頻度とソースコードの更新頻度との間の関係を調査した．コンポーネントテストは，他のテストとは異なり，おおむねソースコードの変更に合わせた修正が行われていることが明らかになった．また，コンポーネントテストを頻繁に更新する必要が要因として，アサーションに HTML を用いていることを挙げ，それを用いずにテストを記述することで，テスト更新の手間を省ける可能性を示唆した．

本研究の貢献は以下の 3 点である．

- フロントエンドフレームワークを採用したプロジェクトにおけるテストの採用状況や品質指標を定量的に明らかにした．
- テストの使用率や更新頻度，テスト失敗理由などに関する課題を特定した．
- テストの改善に向けた具体的なフィードバックを示した．

2 準備

2.1 Web フロントエンドフレームワーク

Web フロントエンドフレームワークとは，ユーザーが直接操作するブラウザ上のインタフェース（フロントエンド）を効率的に開発するためのソフトウェアツールやライブラリの集合体である [1, 2]．これらのフレームワークは，単純な HTML/CSS/JavaScript による開発から進化し，コンポーネントシステムや，リアクティブデータバインディングなど，より高機能でモジュール化された開発を可能にしている．代表的な Web フロントエンドフレームワークには，React, Vue.js, Angular, Svelte などがあり，これらは現在，Web アプリケーション開発の主流となっている．

Web フロントエンドフレームワークに共通する機能としては以下が挙げられる．

- コンポーネントシステム：コンポーネントシステムは，ユーザーインタフェースを再利用可能な単位に分割して構築する手法である．この機能により，コードのモジュール性，再利用性，およびメンテナンス性が向上する．
- データバインディング：アプリケーションのデータ（モデル）と UI（ビュー）を結び付け，双方向または単方向に同期する手法である．これにより，手動で DOM を操作する必要がなくなり，開発者の負担が軽減される．
- 状態管理：アプリケーションの動的な状態を効率的に追跡し，UI とデータの整合性を保つ手法である．

2.1.1 React

React[*1]は，Facebook(現:Meta) によって開発された Web フロントエンドフレームワークである [3]．React のコンポーネントシステム，データバインディング，状態管理の方法を以下に示す．

- コンポーネントシステム：React では JSX を用いてコンポーネントを構築する．JSX は React における UI 構築のための独自の構文拡張であり，JavaScript コード内に，HTML に似た構文を直接記述できる特徴がある．listing 1 中の 11 行目から 16 行目が JSX の記述である．React コンポーネントにはクラスコンポーネントと関数コンポーネントの 2 種類があり，現在は関数コンポーネントが公式に推奨されている．listing 1 は，関数コンポーネントの例である．
- データバインディング：React では単方向バインディングが用いられている．データは親コンポーネントから子コンポーネントのみ流れ，逆方向には流れない．

[*1] https://react.dev/

```jsx
import React, { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };

  return (
    <div style={{ textAlign: "center", marginTop: "50px" }}>
    <h1>Count: {count}</h1>
    <button onClick={increment} style={{ fontSize: "16px", padding: "10px" }}>
    Increase Count
    </button>
    </div>
    );
}
export default Counter;
```

Listing 1: JSX を利用した React の関数コンポーネントの例

- 状態管理：React では状態管理の方法として Hooks が用いられている．Hooks は，React の関数コンポーネントで状態管理やライフサイクルイベントを簡単に扱えるようにする仕組みである．2018 年にリリースされた React v16.8 で導入された．listing 1 中の 4 行目が Hooks の記述である．

2.1.2 Vue.js

Vue.js[*2]は，Evan You によって開発された Web フロントエンドフレームワークである [4]．Vue.js のコンポーネントシステム，データバインディング，状態管理の方法を以下に示す．

- コンポーネントシステム：Vue.js コンポーネントは独自の記法を用いており，template（ビュー），script（ロジック），および style（スタイル）の 3 つの部分から構成される．listing 2 は，これらの 3 つの部分をすべて記述した Vue.js コンポーネントの例である．
- データバインディング：Vue.js には単方向データバインディングと双方向データバインディングの 2 種類がある．単方向データバインディングはユーザー操作によって更新されない要素に，双方向データバインディングは主にフォーム要素（入力フィールド，チェックボックスなど）に用いられる．listing 2 中の 3,5 行目が単方向データバインディング，listing 2 中の 4 行目が双方向データバインディングの記述である．
- 状態管理：Vue.js では状態管理のために，Composition API が用いられる．listing 2 中の 12 行目から 15 行目は Composition API における ref 関数を用いて状態を管理している例である．

[*2] https://vuejs.org/

```vue
<template>
  <div class="container">
  <h1>{{ message }}</h1>
  <input v-model="inputText" type="text" placeholder="テキストを入力">
  <p>入力内容: {{ inputText }}</p>
  <button @click="increment">カウント: {{ count }}</button>
  </div>
</template>

<script setup>
import { ref } from 'vue';
  const message = ref("Hello");
  const inputText = ref("");
  const count = ref(0);
  const increment = () => {
    count.value++;
  };
</script>

<style scoped>
  .container {
    max-width: 400px;
    margin: auto;
    text-align: center;
    font-family: Arial, sans-serif;
  }
</style>
```

Listing 2: Vue.js コンポーネントの例

2.1.3 Angular

Angular[*3]は，Google によって開発された Web フロントエンドフレームワークである [5]．Angular のコンポーネントシステム，データバインディング，状態管理の方法を以下に示す．

- コンポーネントシステム：Angular ではテンプレート構文を利用したコンポーネントシステムが用いられる．`@Component` デコレータを用いてコンポーネントを定義し，HTML 要素を記述できる．listing 3 中の 4 行目から 11 行目がテンプレート構文にあたる．
- データバインディング：Angular では，Vue.js と同様に，単方向データバインディングと双方向データバインディングが用いられる．listing 3 中の 9 行目が単方向データバインディング，listing 3 中の 8 行目が双方向データバインディングの記述である．
- 状態管理：親子コンポーネント間でデータを相互に受け渡すことによって状態を管理する．

[*3] https://angular.dev/

```
1  import { Component, NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3  import { FormsModule } from '@angular/forms';
4  @Component({
5    selector: 'app-root',
6    template: `
7    <h1>Simple Form Example</h1>
8    <input [(ngModel)]="name" placeholder="Enter your name" />
9    <p>Hello, {{ name || 'Guest' }}!</p>
10   `,
11 })
12 export class AppComponent {
13   name = '';
14 }
15 @NgModule({
16     declarations: [AppComponent],
17     imports: [BrowserModule, FormsModule],
18     bootstrap: [AppComponent],
19 })
20 export class AppModule {}
```

Listing 3: Angular コンポーネントの例

2.1.4 Svelte

Svelte[*4]は，Rich Harris によって開発された Web フロントエンドフレームワークである．[6] Svelte のコンポーネントシステム，データバインディング，状態管理の方法を以下に示す．

- コンポーネントシステム：Svelte ではコンポーネントを定義するために独自の記法をが用いられる．コンポーネント定義ファイルである `.svelte` ファイル内には HTML, CSS, JavaScript を結合して記述できる．listing 4 は，これら 3 つをすべて記述した Svelte コンポーネントの例である．
- データバインディング：Svelte では，Vue.js と同様に，単方向データバインディングと双方向データバインディングが用いられる．listing 3 中の 19 行目が単方向データバインディング，listing 3 中の 20,21 行目が双方向データバインディングの記述である．
- 状態管理：Svelte の機能の 1 つである Store には個々のコンポーネントに属さないデータを置くことができ，これを用いて状態を管理できる．listing 3 中の 3 行目では，Store に保持されている変数 `count` をインポートしている．

2.2 Web フロントエンドフレームワークにおけるテスト

Web フロントエンドフレームワークには，ユーザーインタフェースの動的な挙動や，データバインディングや状態管理を含む複雑なコンポーネント構造が存在するため，信頼性の高い動作を保証するた

[*4] https://svelte.dev/

```svelte
<script>
  import Child from './Child.svelte';
  import { count } from './store.js';

  let message = "Hello";
  let userInput = "";
</script>

<style>
  .container {
    max-width: 400px;
    margin: auto;
    text-align: center;
    font-family: Arial, sans-serif;
  }
</style>

<div class="container">
<h1>{message}</h1>
<input bind:value={userInput} type="text">
<p>Input: {userInput}</p>
<Child />
</div>
```

Listing 4: Svelte コンポーネントの例

めにはテストが欠かせない．主に以下の種類のテストが活用される．

- **単体テスト：** 個々の関数やコンポーネントの動作を検証する．
- **結合テスト：** 複数の関数やコンポーネント間の連携や，状態管理の正確性を確認する．
- **E2E テスト（エンドツーエンドテスト）：** ユーザーインタラクションのシミュレーションを通じて，アプリケーション全体の動作を確認する．

2.2.1 単体テスト・結合テスト

ソフトウェアテストの分野において広く知られている概念として，単体テストと結合テストが存在する．一般的に，単体テストは単一の機能を検証するテスト，結合テストは複数の機能の組み合わせを検証するテストと定義される [7, 8]．

しかしながら，単体テスト・結合テストをどこで切り分けるかについては，明確な定義が存在しない．フロントエンドフレームワークにおいても，その定義は曖昧である．よって，本研究ではこれらを区別することは困難であると判断したため，単体テストと結合テストをひとまとめにして扱う．

フロントエンドフレームワークに特有の単体・統合テスト手法として，コンポーネントを対象したテストがある．コンポーネントを対象としたテストでは，コンポーネントの入力や操作から得られるレンダリング結果を検証することで，コンポーネントの正常な動作を確認する．コンポーネントを対象としたテストはフロントエンドフレームワークおいて重要視されている．例えば Vue.js 公式のテストガイ

ド[*5]では，「Vue アプリケーションの多くはコンポーネントテストでカバーされるべきである」と記述されている．

コンポーネントを対象としたテストでは，スナップショットテストと呼ばれる手法がよく用いられる．スナップショットテストではコンポーネントの出力（UI の見た目）が想定外に変更されていないかを検証するテストである．初回のテスト実行時にはアサーションが実行されず，コンポーネントの出力（HTML やスクリーンショット）がファイルに保存される．2 回目以降の実行時にファイルとコンポーネントの出力が照合され，アサーションが行われる．スナップショットテストは予期せぬ UI 変更の検知に対して有効である一方，それ以外の目的で使用することは推奨されていない．例えば Vue.js 公式のテストガイドでは，「スナップショットテストだけに頼るべきではなく，意図を持ってテストを書くべきである」と記述されている．

単体テストや結合テストはフレームワークやライブラリを用いずに実施可能であるが，大多数のプロジェクトではテストフレームワークやテストライブラリが用いられる．単体テストおよび結合テストで使用される主なテストフレームワークやテストライブラリを 表 1 に示す．なお，表 1 中の「推奨」列には，そのテストフレームワークを公式ドキュメントにおいて推奨しているフロントエンドフレームワークを記載している．

2.2.2 E2E テスト（エンドツーエンドテスト）

E2E テスト（エンドツーエンドテスト）とは，実際のブラウザ環境でユーザーインタラクションのシミュレーションを通じて，アプリケーション全体の動作を確認するテスト手法である [9]．E2E テストは，主に以下のような流れでアプリケーションの動作を検証する．

- ブラウザでテスト対象のページを開く．
- ロケータを用いて UI 要素を取得する．なお，ロケータとは，ページ内の要素を ID やクラス，カ

表 1: 単体テスト・結合テストで使用される主なテストフレームワーク・テストライブラリ

名称	種別	URL	推奨
Jest	フレームワーク	`https://jestjs.io/`	React
Vitest	フレームワーク	`https://vitest.dev/`	Vue.js, Svelte
Ava	フレームワーク	`https://github.com/avajs/ava`	
Mocha	フレームワーク	`https://mochajs.org/`	
Jasmine	フレームワーク	`https://jasmine.github.io/`	Angular
Testing Library	ライブラリ	`https://testing-library.com/`	

[*5] https://ja.vuejs.org/guide/scaling-up/testing

スタム属性などを用いて選択するための機能である．

- 取得した UI 要素に対して，クリック，入力などのユーザー操作を実行する．
- 操作の結果，正しい挙動（画面遷移や HTML 要素の変化）が起きているかを確認（アサート）する．

なお，E2E テストのアプローチとして，ユーザー操作をキャプチャして，テスト時に再生するという手法も存在する．しかし，この手法はロケータを用いたプログラマブルな手法よりも保守のコストが多くかかるため，現在ではあまり用いられていない [10]．

E2E テストの目的は，すべてのシステム間の連携や，実際のブラウザ環境におけるユーザー操作が期待とおりに動いているかどうかを検証することである．これは，単体テストや結合テストの目的とは大きく異なる．また，テスト手法に関しても単体テストや結合テストとは明確な区別が可能であるため，本研究では E2E テストを，それらとは別に扱う．

E2E テストで実際のユーザーインタラクションをシミュレートするためには，それに特化したテストフレームワークやテストライブラリが必要になる．E2E テストで使用される主なテストフレームワークを 表 2 に示す．なお，単体テスト・結合テストで使用されるテストフレームワークやテストライブラリと，E2E テストで使用されるテストフレームワークを組み合わせてテストが行われることもある．

2.2.3　テストの品質指標

Web フロントエンドフレームワークにおけるテストには，他のソフトウェア開発におけるテストと同様に，いくつかの品質指標が存在する．

品質指標の 1 つにテストの成功率がある．テストの成功率は，テストスイート全体のテストケースのうち，どれだけの割合が成功したかを示す指標である．テストの成功率が高いほど，テストスイートが正常に動作していることを示し，ソフトウェアの品質向上に寄与する．失敗するテストが存在すること自体が問題であるため，テストの成功率が 100% であることが望ましい．なぜなら，失敗するテストが

表 2: E2E テストで使用されるテストフレームワーク・テストライブラリ

名称	種別	URL
Cypress	フレームワーク	https://www.cypress.io/
Playwright	フレームワーク	https://playwright.dev/
Puppeteer	ライブラリ	https://pptr.dev/
Selenium WebDriver	ライブラリ	https://www.selenium.dev/documentation/webdriver/
Nightwatch.js	フレームワーク	https://nightwatchjs.org/
Karma	フレームワーク	https://karma-runner.github.io/

存在すると，そのテストが真にソースコードの問題を示しているのか，テスト自体に問題があるのか判断するコストが発生するからである．しかしながら，実際には様々な理由で失敗するテストが存在し，テストの成功率が 100% にならない場合がある．

また，カバレッジも品質指標の 1 つとして利用される [11, 12]．カバレッジは，ソフトウェアテストにおいて，テストがソースコードをどの程度を網羅しているかを測定する指標である．具体的には，テストスイートがアプリケーション内のどの部分を実行し，どの部分を実行していないかを定量的に評価することで，テストの網羅性を把握するために使用される．カバレッジは，テストの品質や有効性を評価するための重要な指標であり，十分なカバレッジを確保することで，未検証のコード領域が減少し，潜在的なバグが検出される可能性が高まるため，ソフトウェアの信頼性と品質向上を目指すことができる．

カバレッジの種類には，ステートメントカバレッジ，ブランチカバレッジ，関数カバレッジ，条件カバレッジなどがあり，いずれも Web フロントエンドフレームワークにおけるテストで使用される．

2.3 オープンソースソフトウェア (OSS)

オープンソースソフトウェア (OSS) は，ソフトウェアのソースコードが公開されており，誰でも自由に利用，改変，再配布できるライセンスが付与されたソフトウェアを指す [13]．これにより，OSS は単なるソフトウェアの提供に留まらず，グローバルな共同開発や知識の共有を可能にするプラットフォームとして機能している．

OSS は，そのソースコード，開発履歴，テストコードなどが GitHub や GitLab などのプラットフォーム上で公開されており，研究者にとってデータを取得しやすい環境を提供する．その中でも GitHub は，マイニング研究の対象として盛んに利用されているプラットフォームであり，API などを用いてリポジトリの情報を取得し，オープンソースソフトウェアからデータセットを構築する手法が確立されている [14]．

3 Research Questions

本研究では，Web フロントエンドフレームワークを採用したオープンソースプロジェクトにおけるテストの現状と課題を調査する．テストの現状と課題について多角的な視点から調査するため，以下に示す 4 つの Research Questions を設定した．なお，RQ1, RQ3 については，様々な言語のオープンソースプロジェクトにおけるテストについて調査した Kochhar らの先行研究 [15] で設定された RQ を参考にしている．

RQ1: Web フロントエンドフレームワークを採用したプロジェクトにおけるテストの使用率は？

RQ1 では，Web フロントエンドフレームワークを採用したプロジェクトにおける各種のテスト（単体・結合テストおよび E2E テスト，コンポーネントテスト）の使用率を調査し，これらのテストがどの程度実施されているかを明らかにする．また，コンポーネントテストのうち，スナップショットテストが占める割合を調査することにより，コンポーネントテストがフレームワークのテストガイドで推奨される方法で行われているかどうかを確かめる．具体的には，`package.json` に記述された依存関係やテストコマンドをもとに，各プロジェクトで各種のテストを実行するためのスクリプトが定義されており，かつそれが実行可能かどうかを判定することにより，使用率を算出する．コンポーネントテスト，スナップショットテストについては，実際のソースファイルの記述をもとに，使用の有無を判断する．なお，先述した通り，本 RQ は Kochhar らの先行研究を参考に設定している．よって，Kochhar らの先行研究と本 RQ の調査結果を比較することにより，フロントエンドフレームワークを採用したプロジェクトは，他の種類のプロジェクトよりもテストの使用率が高いかどうかを判定する．

RQ2: Web フロントエンドフレームワークを採用したプロジェクトにおけるテストの品質は？

RQ2 では，テストの品質に関わる指標としてテストの成功率・カバレッジの 2 つをプロジェクトごとに算出する．これらの品質指標を得るため，Web フロントエンドフレームワークを採用した各プロジェクトで単体・結合テストおよび E2E テストを実行する．また，テストのカバレッジについては，コンポーネントとそれ以外のファイルに分けて集計することで，コンポーネント定義ファイルが十分にテストされているかどうかを明らかにする．さらに，失敗テストの原因やカバレッジが低いステートメントの種類を特定することで，テストの品質に悪影響を及ぼす要因を明らかにする．

RQ3: Web フロントエンドフレームワークを採用したプロジェクトにおいて，バグ報告数とテストケースの数・品質等との関係は？

RQ3 では，RQ1 で調査したテストケースの数，および RQ2 で調査したテストのカバレッジと，GitHub 上におけるバグ報告数との関係を調査する．なお，プロジェクトの規模が大きくなるに従ってテストケースの数やバグ報告数は大きくなる傾向にあるため，これらの値についてはプロジェクト全体の LOC（コード行数）で割って正規化した値を調査に用いる．これにより，テストケースの数やテスト

の品質がバグ報告数にどのような影響を与えるかを明らかにする．

RQ4: Web フロントエンドフレームワークを採用したプロジェクトにおけるテストの更新頻度や，ソースコードの更新頻度との関係は？

RQ4 では，GitHub におけるコミット履歴をもとに，ソースコード更新に対するテストコード更新の相対頻度や，ソースコード更新頻度とテストの更新頻度との関係について調査する．なお，テストの更新頻度については，コンポーネントテストとそれ以外のテストで分けて集計を行う．これにより，コンポーネントテストの修正頻度が他の種類のテストと比べて高いかどうかや，ソースコードの修正に伴いテストコードも共に更新される傾向にあるかどうかを明らかにする．

4　調査手法

本研究における調査は，以下に示す流れに従って進める．

1. リポジトリの収集
2. リポジトリのフィルタリング
3. テストの実行
4. テストコード・実行結果の抽出
5. 特定の RQ に関する調査
    - 特徴のあるステートメントのカバレッジに関する調査 (RQ2)
    - テスト数やカバレッジとバグ報告 Issue との間の相関関係の調査 (RQ3)
    - テスト更新頻度の調査 (RQ4)

調査の大まかな流れを 図 1 に示す．また，リポジトリをフィルタリングするフローの詳細を 図 2 に示す．

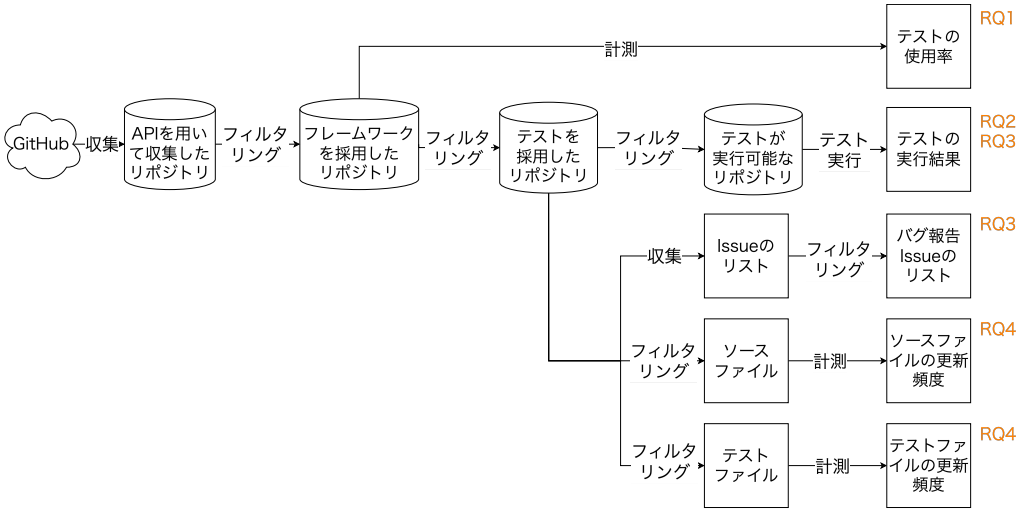


図 1: 調査の大まかな流れ

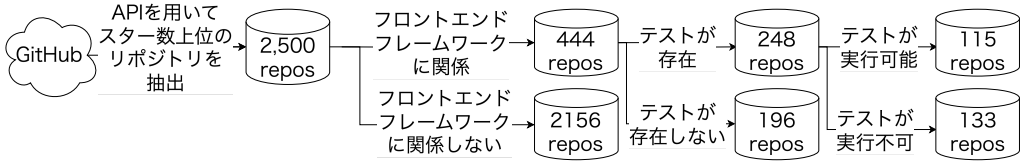


図 2: リポジトリをフィルタリングする詳細なフロー

4.1 リポジトリの収集

リポジトリの収集元には，マイニング研究において盛んに利用されており，API を用いてデータが収集しやすい GitHub を選択した．

Web フロントエンドフレームワークを用いたプロジェクトでは，プログラミング言語として JavaScript, TypeScript が主に使用される．また，Vue.js, Svelte のコンポーネントはそれぞれ `.vue`, `.svelte` という拡張子を持つため，GitHub においては他の言語と区別されている．また，GitHub におけるスター数は，プロジェクトの人気度を示す指標として広く利用されており，スター数が多いプロジェクトほど，多くの開発者によってメンテナンスされている可能性が高い．よって，本研究では検索条件として言語を JavaScript, TypeScript, Vue.js Svelte のいずれかに絞り，その条件に当てはまるスター数上位 2,500 件のプロジェクトを，GitHub API を用いて収集した．なお，これらの中には Web フロントエンドフレームワークを採用していないプロジェクトが多く存在するため，以降の段階でフィルタリングが必要である．

4.2 リポジトリのフィルタリング

収集したプロジェクトの中から，Web フロントエンドフレームワークを採用したプロジェクトのみを抽出するため，以下のすべての条件を満たすプロジェクトのみをフィルタリングした．

- プロジェクトのトップレベルに `package.json` が存在する．なお，`package.json` は，Node.js のパッケージマネージャーである npm で使用される，プロジェクトの依存関係を記載したファイルである [16].
- モノレポではない．
  - モノレポとは，1 つのプロジェクトで複数のパッケージを管理する方法の 1 つである [17].
  - モノレポを採用したプロジェクトでは，複数のパッケージを管理する都合上，Web フロントエンドフレームワークに関係しないパッケージを多く含むため，本研究の調査対象からは除外した．
- `package.json` の中に，依存関係として Web フロントエンドフレームワークに関するパッケージを含む．
  - Web フロントエンドフレームワークに関するパッケージとは，`react`, `vue`, `angular`, `svelte` のいずれかである．
  - `dependencies` と `devDependencies` フィールドの両方を依存関係収集の対象とする．
- コンポーネントを定義したファイルをソースコード中に含む．
  - 依存関係として Web フロントエンドフレームワークに関するパッケージを含むが，コン

ポーネントが存在しないプロジェクトは Web フロントエンドフレームワークを使用しているプロジェクトとは言えないため，本研究の調査対象から除外した．

  - コンポーネントを定義しているファイルは他のファイルと違い，コンポーネント特有の構文を用いている．このことから，プロジェクトのソースコードに特有の構文を含むファイルが存在するかどうかをパーサーを用いて検証し，フィルタリングを行った．

- ネイティブアプリをターゲットとしていない．
  - Web フロントエンドフレームワークと類似した方法を用いてネイティブアプリを開発する手法として，React Native などがある．
  - ネイティブアプリをターゲットとしているプロジェクトには，ターゲット OS に合わせた特殊な処理が含まれており，通常の Web フロントエンドフレームワークと単純に比較することが困難である．よって，このようなプロジェクトは本研究の調査対象から除外した．

収集したプロジェクトのうち，フィルタリングを通過したプロジェクトは 444 件であった．これらのプロジェクトにおいて，各フレームワークの使用数を表したオイラー図を 図 3 に示す[*6]．図 3 から分かるように，本調査で収集したプロジェクトの多くは React または Vue.js を使用しており，Angular, Svelte を使用したプロジェクトは少数であることがわかる．

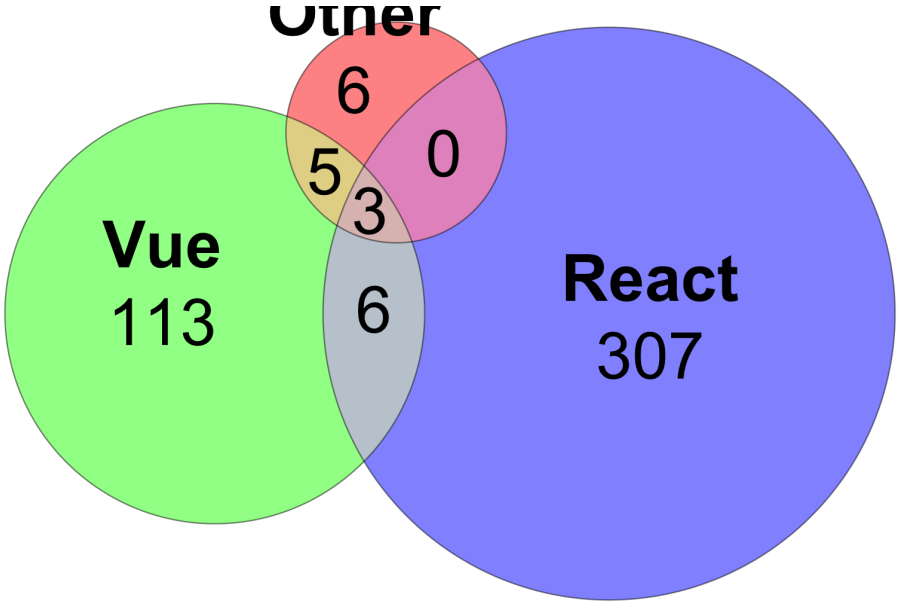


図 3: 各フロントエンドフレームワークの採用プロジェクト数を表すオイラー図

[*6] 「その他」には，Angular, Svelte を含む．

4.3 テストの実行

RQ2 の調査に必要なテストの品質指標（テストの成功率・カバレッジ）を得るため，実験対象プロジェクトに含まれるすべてのテストを実行した．

テストを実行する前に，テストがプロジェクトに存在するかを判定する必要がある．よって，以下の条件を立て，これらをすべて満たすプロジェクトを，テストが実行可能であるプロジェクトであると判定した．

- パッケージマネージャーを用いた依存関係のインストールが正常に終了する．
  - 「インストールが正常に終了する」とは，「パッケージマネージャーにおけるデフォルトのインストールコマンドを実行した際に，前処理や後処理を含めて，すべての段階の処理が正常に終了する」という意味である．
  - インストールに失敗した場合，仮にプロジェクト内にテストが存在している場合であっても，この時点でテスト実行の対象外とする．
- `package.json` の中に，依存関係としてテストフレームワークに関するパッケージを含む．
  - 表 1 及び 表 2 に記載されているテストフレームワークが対象である．
  - `dependencies` と `devDependencies` フィールドの両方を依存関係収集の対象とする．
- `package.json` の`"script"`フィールドに，テストを実行するためのスクリプト名が記述されており，なおかつスクリプトと対応するコマンドを実行することで，テストが実際に実行される．
  - `package.json` の`"script"`フィールドには，例えば`"test": "jest"`のように，スクリプト名と，実際に実行さるコマンドの組が列挙されている．
  - 表 3 は，テストの種類と，対象にするテストスクリプトを検索するための正規表現との対応を表している．例えば，スクリプト名に `unit` という文字列が含まれている場合，単体テストを実行するスクリプトであると判定する．
  - 表 3 の「種類不明のテスト」に対応するスクリプト名は，それを見ただけでは種類が判別できないスクリプト名を表している．これらのスクリプトが `package.json` 内に存在する場合，対応するコマンドを実際に見て，テストの種類を判定した．
  - 「スクリプトと対応するコマンドを実行することで，テストが実際に実行される」という条件により，例えば`"test": "echo 'no tests.'"`のように，スクリプトを実行しても実際にはテストが実行されない場合を除外できる．

この手順により，テストの実行が可能であると判断したプロジェクトは合計で 115 件であった．

実行可能であると判断したテストについては，以下の手順に基づいて実行した．

- 単体・結合テストについては，json でのテスト結果の取得，およびカバレッジの取得（Jest,Vitest のみ）のため，使用しているテストフレームワークに合わせてコマンドを一部改変し，自動で実行した．
  - テストフレームワークごとのコマンドの改変方法を 表 4 に示す．
  - 改変後のコマンドに記載している `[options]` は，テストスクリプトに記載されているオプションのうち，追加したオプションと競合しないオプションを指す．例えば，テストスクリプトに `jest --timeout=10000 --json` と記載されている場合，`[options]` には`--timeout=10000` が含まれる．
- E2E テストについては，Readme の内容などを元に，テストの動作に必要な操作をすべて手動で実行した後，テストコマンドを手動で実行した．なお，テストの動作に必要な操作とは，ビルドや API サーバーの起動，ブラウザのインストールなどである．

4.4 テストコード・実行結果の抽出

テストをすべて実行した後には，テストの実行ログなどを解析し，表 5 に示すような指標を取得した．なお，一部のテストフレームワーク (Mocha, Jasmine, Ava と，すべての E2E テストフレームワーク) を採用したプロジェクトにおいてはカバレッジの取得が困難であった．よって，4.3 節にも記述したように，カバレッジの取得対象となったのは，Jest, Vitest を採用したプロジェクトのみである．

表 3: テストの種類と，それに対応するテストスクリプトを検索するための正規表現

テストの種類	対象にするテストスクリプトを検索するための正規表現
単体テスト	`.*unit.*`, `.*jest.*`, `.*vitest.*`, `.*jasmine.*`
結合テスト	`.*integration.*`
E2E テスト	`.*e2e`, `.*browser.*`, `.*cypress.*`, `.*puppeteer`, `.*selenium.*`, `.*chrome.*`, `.*firefox.*`, `.*webkit.*`,`.*electron.*`
種類不明のテスト	`test`, `test:run`, `test:all`

表 4: 各単体・結合テストフレームワークにおける改変後のコマンド

テストフレームワーク	改変後のコマンド
jest	`jest --coverage --json [options]`
vitest	`vitest --run --coverage --reporter=json [options]`
ava	`ava --tap [options] | tap-json > report.json`
jasmine[*7]	`jasmine [options] > report.txt`
mocha	`mocha --reporter=json [options]`

4.5 特徴のあるステートメントのカバレッジに関する調査

非同期処理や，コンポーネントの記述などの特徴のあるステートメントは，カバレッジが低くなる傾向にあると考えられる．よって，RQ2 の調査では，これらのステートメントについては個別に統計を取り，全体のステートメントカバレッジの平均値と比較することにより，カバレッジが実際に低い傾向にあるかどうかを確かめる．カバレッジ統計の対象となるステートメントの種類と，それを設定した理由を 表 6 に示す．

4.6 テスト数やカバレッジとバグ報告 Issue との間の相関関係の調査

RQ3 の対象であるテスト数やカバレッジとバグ報告 Issue との間の相関関係については，以下の手法で調査を行う．

- GitHub API を用いて，リポジトリの Issue 一覧と contributor 一覧を取得する．
- Issue のラベルとタグを元に，バグ報告 Issue をフィルタリングする．なお，本研究では，タイトルまたはタグに “bug”, “bugs”, “buggy”, “defect”, “defects”, “error”, “errors” のいずれかの

表 5: 本研究における調査のために取得する指標

指標	取得方法	関連する RQ
各種テスト（単体・結合，E2E, コンポーネントテスト）のテストケース数	テスト実行時のログ，ファイル名，ファイルの記述から	RQ1 ,RQ2, RQ3
コンポーネントテストに占めるスナップショットテストの割合	ファイルの記述から	RQ1
テストの成否	テスト実行時のログから	RQ2
カバレッジ	テスト実行時に出力される `coverage-final.json` から	RQ2, RQ3

表 6: カバレッジ統計の対象となるステートメントの種類

ステートメントの種類	設定理由
外部リソースのや API の fetch	外部リソースや API は変更されることがあり，テストしにくい
Try-Catch 節	エラーパスがテストされにくい傾向にある
React の Hook	コンポーネントのテスト手法が特殊
JSX のイベントハンドラ	コンポーネントのテスト手法が特殊
JSX の制御構文	コンポーネントのテスト手法が特殊
Vue.js の Composition API	コンポーネントのテスト手法が特殊

文字列を含む Issue をバグ報告 Issue と定義する．

- リポジトリごとにバグ報告 Issue の数とテストの数を算出し，それらの相関関係を統計ソフトを用いて調査する．データは正規分布に従わないと仮定できるため，ノンパラメトリック検定であるスピアマンの順位相関係数を用い，相関関係を判定する．

4.7 テスト更新頻度の調査

RQ4 の対象であるテスト更新頻度については，以下の手法で調査を行う．

- プロジェクト中のファイルをコンポーネントテスト，コンポーネントテスト以外の単体・結合テスト，コンポーネント定義ファイル，コンポーネント定義ファイル以外のソースの 4 種類に分ける．なお，テストコードからインポートされていないソースコードはテストの更新に影響を与えないため，本調査の対象から除外する．
- それぞれ Git の更新履歴を元にテストの更新頻度を算出する．更新頻度は，コミットログにおけるテストファイルの更新回数を，そのファイルの生存期間（日単位）で割ることで算出する．
- (コンポーネントテストの更新頻度 / コンポーネント定義ファイルの更新頻度)，および (コンポーネントテスト以外の単体・結合テストの更新頻度 / コンポーネント定義ファイル以外のソースの更新頻度) をそれぞれ求める．
- ソースファイルの更新頻度とテストファイルの更新頻度との相関関係を，統計ソフトを用いて調査する．データは正規分布に従わないと仮定できるため，ノンパラメトリック検定であるスピアマンの順位相関係数を用い，相関関係を判定する．

5　調査結果と考察

5.1　RQ1

5.1.1　結果

フロントエンドフレームワークを採用したプロジェクトにおいて，単体・結合テスト，E2E テスト，コンポーネントテストの採用数を表したオイラー図を 図 4 に示す．何らかのテストを採用しているプロジェクトは，全体のうち 55.18% であった．また，コンポーネントテストを採用しているプロジェクト数は，単体・結合テストを採用しているプロジェクト数の約 24.9%，E2E テストを採用しているプロジェクト数は，単体・結合テストを採用しているプロジェクト数の約 22.0% であった．これらより，単体・結合テストの採用数に比べ，コンポーネントテストや E2E テストの採用率が低いことが分かる．

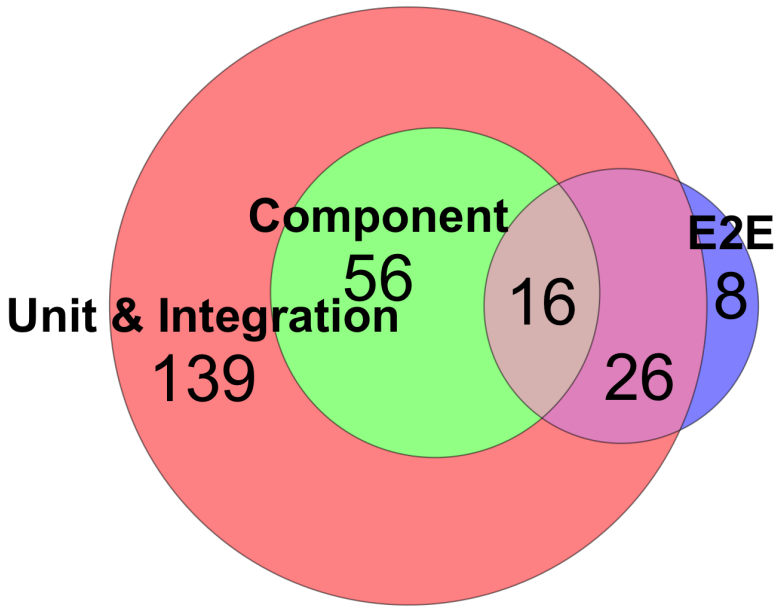


図 4: 単体・結合テスト，E2E テスト，コンポーネントテストの採用数を表すオイラー図

コンポーネントテストのテストケースのうち，スナップショットテストが占める割合を 図 5 に示す．スナップショットテストを使用しているプロジェクトは，コンポーネントテストを採用しているプロジェクトのうち 31.7% だった．このことから，コンポーネントテストのアサーション手法として，スナップショットテストを用いているプロジェクトは少ないことがわかる．

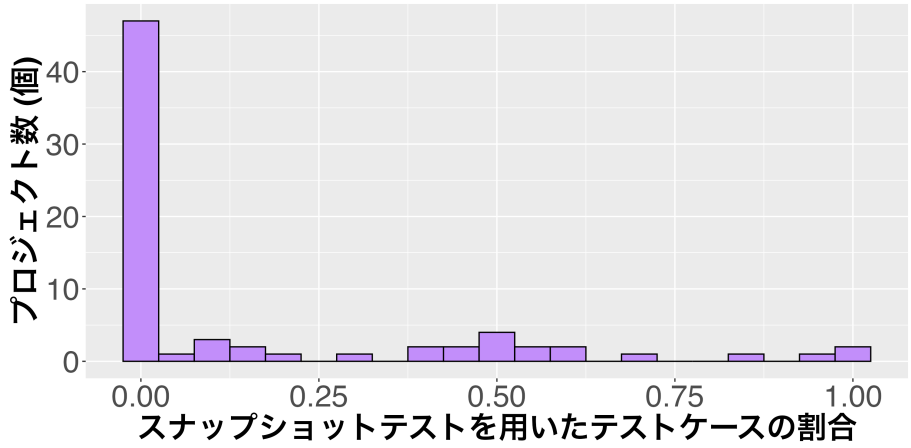


図 5: コンポーネントテストのテストケースのうち，スナップショットテストが占める割合を示したヒストグラム

5.1.2 考察

様々な言語のオープンソースプロジェクトにおけるテストについて調査した Kochhar らの先行研究 [15] では，テストを採用しているプロジェクトは全体のうち 61.65% であった．本研究においては，テストを採用しているプロジェクトは全体のうち 55.18% であった．また，Kochhar らの先行研究においては，テストを採用しているプロジェクトのほうが，採用していないプロジェクトよりもプロジェクトの規模 (開発者数や LOC) が大きい傾向にあることが報告されている．本研究ではスター数が多く，規模が大きい傾向にあるプロジェクトを調査の対象としている一方，Kochhar らの先行研究においては，大小様々な規模のプロジェクトを調査している．よって，仮に規模が大きいプロジェクトのみを Kochhar らの研究の対象とすると，調査結果はより高い値になると考えられる．よって，Web フロントエンドフレームワークを採用したプロジェクトは，他の種類のプロジェクトよりもテストの使用率が低いと言える．

コンポーネントテストを採用しているプロジェクト数は，単体・結合テストを採用しているプロジェクト数の約 1/4 であった．このことから，コンポーネントテストを行っていないプロジェクトが多いことが分かる．このことから，コンポーネントテストに関しては，テストの導入に何らかの障壁があると考えられる．一方，コンポーネントテストテストを行っているプロジェクトのうち，スナップショットテストを採用しているプロジェクトの割合は少なかった．このことから．コンポーネントテストテストを行っているプロジェクトにおいては，概ね公式のテストガイドで推奨される方法を用いてテストを行っていることが分かる．

E2E テストは単体・結合テストを採用しているプロジェクトの数は，単体・結合テストを採用してい

るプロジェクト数の約 22.0% であった．採用数が低い理由として，E2E テストは単体・結合テストと違い，API サーバーなど，バックエンドと一体となってテストが行われることが多いためであると考えられる．具体的には，E2E テストは，アプリケーション全体，特にフロントエンドとバックエンドの両方が連携した全体的な動作を確認することを目的としているため，バックエンドを持たないプロジェクトでは単体テストや結合テストで十分であると判断される場合が多く，採用率が低くなっていると考えられる．

これらの考察から得られるフロントエンドフレームワークにおけるテストの課題については，以下が考えられる．

- テストの実施率が他の種類のプロジェクトよりも低い．
- 特にコンポーネントテストを実施しているプロジェクトが少なく，コンポーネントテストの導入や運用に何らかの障壁があることが考えられる．

5.2 RQ2

5.2.1 結果

プロジェクトごとの単体・結合テストの成功率，および E2E テストの成功率を表すヒストグラムを 図 6 に示す．テストの成功率が 100% であるプロジェクトの割合は，単体・結合テストでは 84.3%，E2E テストでは 65.8% であった．よって，E2E テストは，単体・結合テストよりも成功率が 100% ではないプロジェクトの割合が高いことが分かる．

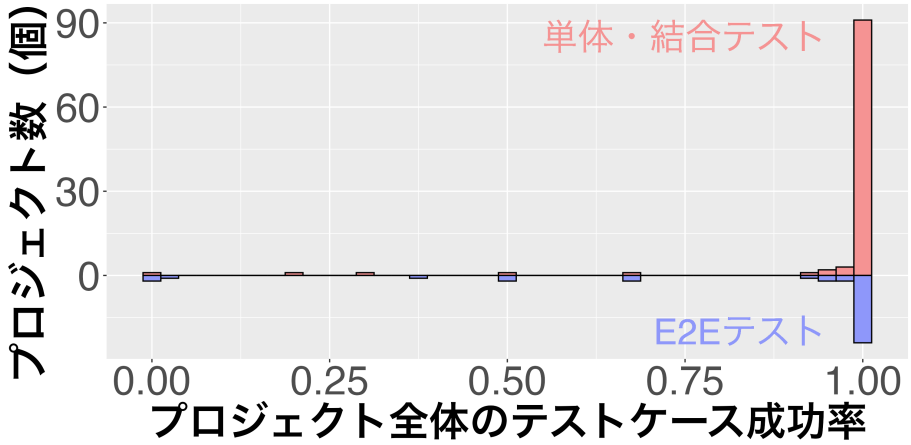


図 6: プロジェクトごとの単体・結合テストの成功率，および E2E テストの成功率を表すヒストグラム

テスト失敗理由の一覧と，それに当てはまるテストケースの数を単体・結合テストおよび E2E テストごとに集計した結果を 表 7 に示す．なお，表 7 中において*印で示した失敗理由はフロントエンドに特

有の実行時エラーである．表 7 から分かるとおり，アサーションの失敗はテスト失敗の主要な要因ではなく，他の理由，特にフロントエンドに特有の実行時エラーで失敗しているテストが多いことが分かる．

プロジェクトごとの単体・結合テストのステートメントカバレッジを表すヒストグラムを 図 7 に示す．また，プロジェクトごとの単体・結合テストのステートメントカバレッジを，コンポーネント定義ファイルとそれ以外のファイルで分けて集計した場合のヒストグラムを 図 8 に示す．単体・結合テストのステートメントカバレッジの平均は 63.53% だった．また，コンポーネント定義ファイルのステートメントカバレッジの平均は 38.30%，その他のファイルのステートメントカバレッジの平均は 65.39% であった．これらより，単体・結合テストのステートメントカバレッジや，コンポーネント定義ファイル以外のカバレッジと異なり，コンポーネント定義ファイルのカバレッジは低く，特にカバレッジが 0 のプロジェクトが多く見られることがわかる．

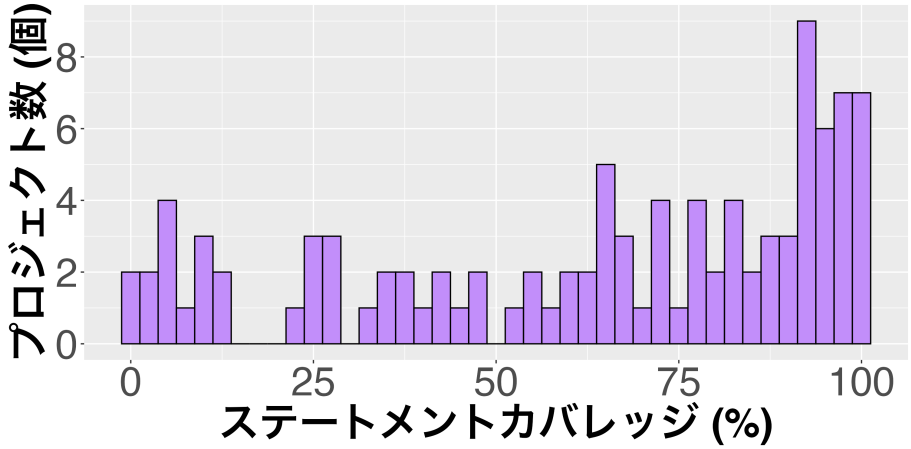


図 7: プロジェクトごとの単体・結合テストのステートメントカバレッジを表すヒストグラム

特徴のあるステートメントのカバレッジをまとめた表を 表 8 に示す．すべてのプロジェクトにおけ

表 7: テストの失敗理由とそれに当てはまるテストケースの数

失敗理由	単体・結合テスト	E2E テスト
mock の設定ミス*	319	0
ネットワーク関連のエラー*	167	7
ロケータの不適切な使用*	86	8
アサーションの失敗	75	0
テストのタイムアウト	59	12
スナップショットの不一致*	12	58
ブラウザのクラッシュ*	0	17
Node.js バージョンの相違	1	0

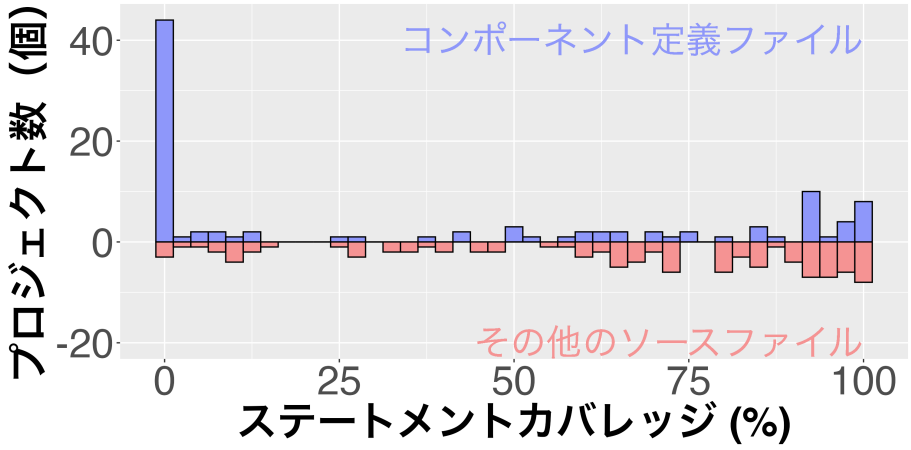


図 8: プロジェクトごとの単体・結合テストのステートメントカバレッジをコンポーネント定義ファイルとそれ以外のファイルに分けて集計したヒストグラム

るカバレッジの平均値は 61.88% であるため，このようなステートメントのカバレッジは平均よりも低いと言える．

5.2.2 考察

5.2.3 テストの成功率に関する考察

E2E テストは，単体・結合テストより成功率が 100% ではないプロジェクトの割合が多かった．よって，E2E テストは単体・結合テストよりも失敗テストが放置されやすい傾向にあると推測される．

表 7 から分かるように，単体・結合テストについては，ソースコードの変更により壊れやすい傾向にあることが知られているロケータ [18] や，mock など，主にフロントエンドフレームワークに特有の理由でテストが失敗していることがうかがえる．また，E2E テストについても，これらに加えてブラウザのクラッシュといった，E2E テスト特有の理由で失敗している．一方，単体・結合テストの失敗理由と

表 8: 特徴のあるステートメントのカバレッジ

ステートメントの特徴	ステートメントカバレッジの平均
外部リソースのや API の fetch	32.9%
Try-Catch 節	46.1%
React の Hook	32.1%
JSX のイベントハンドラ	16.7%
JSX の制御構文	15.7%
Vue.js の Composition API	6.11%

して単純なアサーションエラーが占める割合はあまり高くなく，E2E テストに至っては，単純なアサーションエラーが失敗理由であるテストは存在しなかった．このことから，失敗テストが残っている原因の多くは，フロントエンドフレームワークに特有のテスト手法をうまく扱えていないことであると推察される．

これらの考察から得られるフロントエンドフレームワークにおけるテストの課題については，以下が考えられる．

- E2E テストの失敗テストは放置されやすい．
- フロントエンドフレームワークに特有のテスト手法をうまく扱えておらず，そのせいでテストの失敗が放置されている場合がある．

また，これらの考察より，開発者に対するフィードバックとしては以下が考えられる．

- フロントエンドフレームワークに特有のテスト失敗理由はいくつかあり，それらはテストの保守にとって重要である．

5.2.4　カバレッジに関する考察

2.2.1 節にも記述したように，コンポーネントテストは重要視されている一方，図 8 からわかるように，コンポーネントのカバレッジが 0 のプロジェクトが多く存在する．これは，RQ1 の調査結果から分かる通り，そもそもコンポーネントテストを採用したプロジェクトの数が少ないことが主要な原因であると考えられる．

すべての特徴のあるステートメントのカバレッジは，平均のカバレッジよりも低い傾向にあった．非同期処理やエラーハンドリングのカバレッジが低い原因は，表 6 に示したとおりであると推察される．また，コンポーネントに特有のステートメントのカバレッジが低かったのは，コンポーネントのカバレッジが 0 のプロジェクトが多いことが主な原因であると考えられる．

これらの考察から得られるフロントエンドフレームワークにおけるテストの課題については，以下が考えられる．

- 非同期処理やエラーハンドリング，コンポーネントの記述はテストされにくい傾向にある．

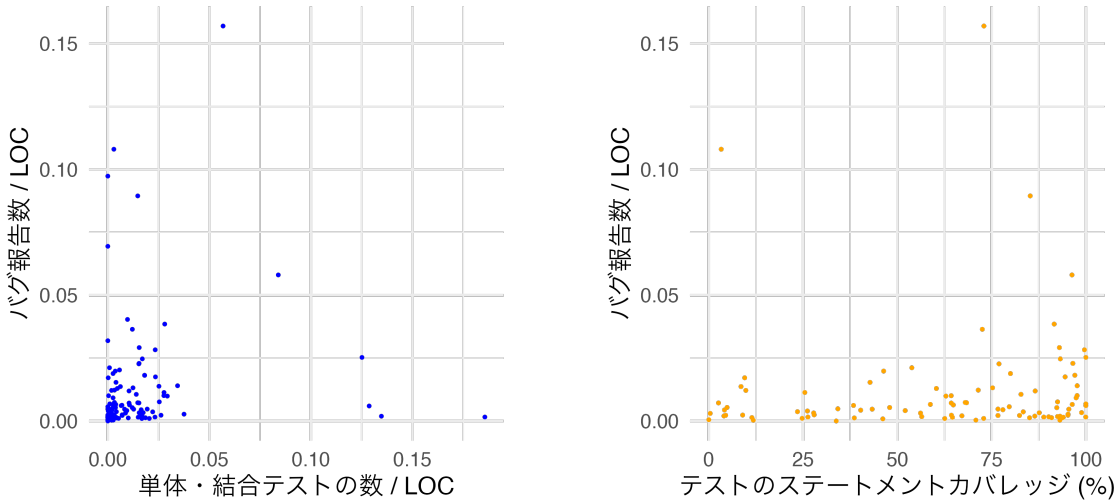
5.3 RQ3

5.3.1 結果

(単体・結合テストテストケース数 / LOC) と (バグ報告数 / LOC) との間の相関を表す散布図を 図 9a に示す．これらの間の相関係数は 0.2791（p 値 = 0.0026）であり，相関はほとんど見られなかった．

単体・結合テストのステートメントカバレッジと (バグ報告数 / LOC) との間の相関を表す散布図を 図 9b に示す．これらの間の相関係数は 0.6044（p 値 = 1.402×10^{-10}）であり，弱い正の相関関係が見られた．

なお，これらの相関を調査した際に，(バグ報告数 / LOC) の値が極端に高い外れ値が 1 つ存在することが確認された．外れ値に該当するリポジトリを参照すると，同じタイトルの Issue が何度も投稿されており，スパムであることが確認された．よって，当該リポジトリは本 RQ の調査対象から外している．



(a) (バグ報告数 / LOC) と (単体・結合テストのテストケース数 / LOC) との間の相関を表す散布図

(b) (バグ報告数 / LOC) と単体・結合テストのステートメントカバレッジとの間の相関を表す散布図

図 9: (バグ報告数 / LOC) と，テストケース数やカバレッジとの間の相関

5.3.2 考察

様々な言語のオープンソースプロジェクトにおけるテストについて調査した Kochhar らの研究 [15] では，バグ報告数とテストの数との間には相関がほとんど見られなかった．本研究においては，(バグ報告数 / LOC) と (テストの数 / LOC) との間には相関がほとんど見られなかった．本研究ではバグ

報告数とテストの数の両方をソースコードの LOC で割って正規化しているため単純に比較はできないが，先行研究と同様，バグ報告数とテストの数との間にはあまり関係がないと考えられる．テストの数が少ないリポジトリのバグ報告 Issue をいくつかサンプリングしてタイトルや内容を見ると，実際にプロジェクトを動かした際に発見したバグや，アプリケーションを使用しているときに発見したバグが多く見られた．よって，フロントエンドフレームワークを採用したプロジェクトにおいては，テストを必要としないバグ報告が多く存在するため，バグ報告数とテストの数との間には関係が見られなかったと考えられる．

一方，(バグ報告数 / LOC) とカバレッジとの間には弱い正の相関関係が認められる．よって，カバレッジが高いほど，LOC に対するバグ報告数が増加する傾向にあることが示唆される．

これらの考察より，開発者に対するフィードバックとしては以下が考えられる．

- バグ報告数を増やすためには，おそらくカバレッジを高めることが重要である．
- バグを見つけるためには，テストのみでは不十分であり，実際にプログラムを動作させることも重要である．

5.4 RQ4

5.4.1 結果

単体・結合テストファイルの更新頻度を，コンポーネントテストとそれ以外のテストで分けて集計したヒストグラムを 図 10 に示す．(テストファイルの更新頻度 / ソースファイルの更新頻度) の平均は，コンポーネントテストでは 0.6258，それ以外のテストでは 0.6194 であった．このことから，ソースコード更新に対するテストコード更新の相対頻度ついては，コンポーネントテストとそれ以外のテストでほとんど差がないことがわかる．

コンポーネントテスト以外のテストファイルの更新頻度と，コンポーネント定義ファイル以外のソースコードの更新頻度との関係を 図 11(11a) に示す．これらの間の相関係数は 0.4307（p 値 $= 0.00015$）であり，非常に弱い相関関係が見られた．また，コンポーネントテストファイルの更新頻度と，コンポーネント定義ファイルの更新頻度との関係を 図 11(11b) に示す．これらの間の相関係数は 0.7547（p 値 $= 2.2 \times 10^{-16}$）であり，やや強い相関関係が見られた．

5.4.2 考察

(テストファイルの更新頻度 / ソースファイルの更新頻度) については，コンポーネントテストとそれ以外のテストでほとんど差がなかった．一方，コンポーネントテストのほうが，ソースコードの更新頻度との相関が強かった．これらより，コンポーネントテスト以外のテストコードがソースコードと共に

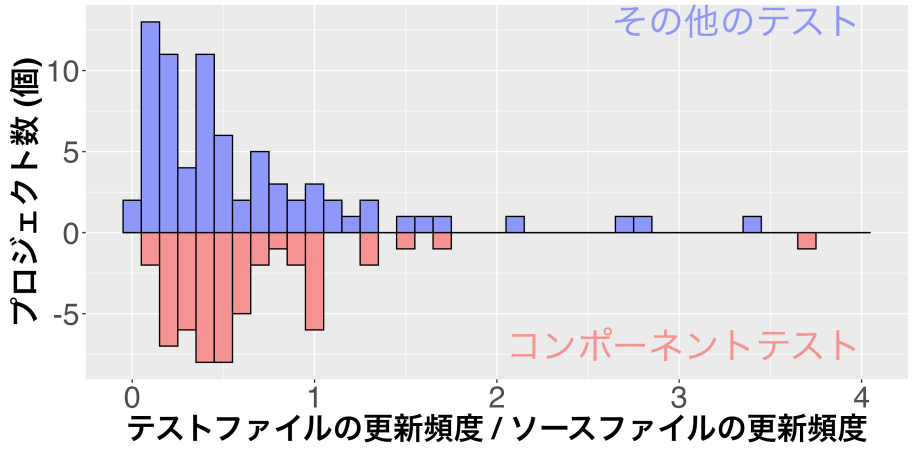
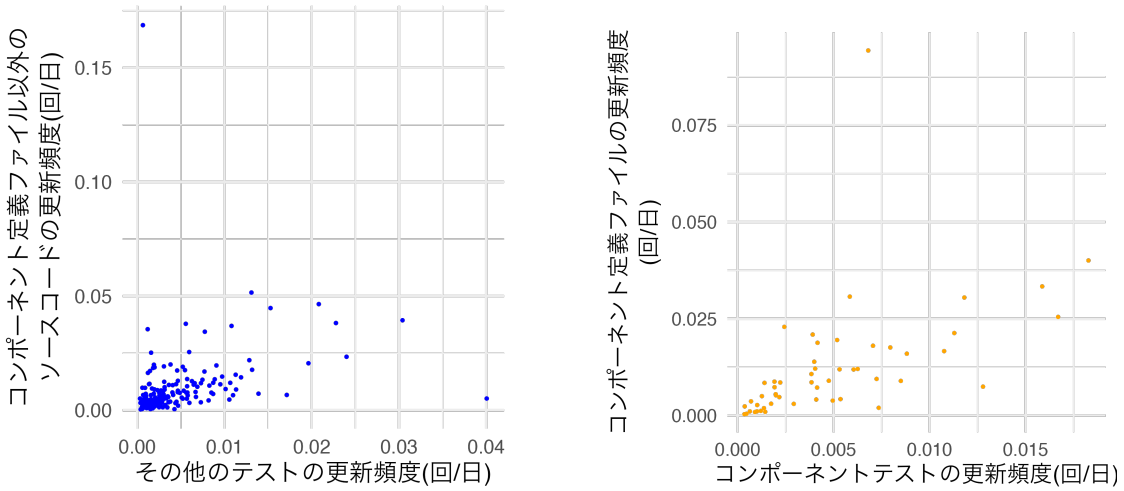


図 10: 単体・結合テストファイルの更新頻度を，コンポーネントテストとそれ以外のテストで分けて集計したヒストグラム



(a) その他のテストの更新頻度とコンポーネント定義ファイル以外のソースコードの更新頻度との相関を表す散布図

(b) コンポーネントテストの更新頻度とコンポーネント定義ファイルの更新頻度との相関を表す図

図 11: テストファイルの更新頻度とソースコードの更新頻度との相関を表す散布図

更新されるかどうかはプロジェクトによってまちまちであるが，コンポーネントテストは，多くのプロジェクトにおいて，ソースコードの更新に合わせて修正される傾向にあることがわかる．

また，頻繁な更新がなされているコンポーネントテストファイルをいくつか抽出してソースコードを見ると，出力のアサーションを行う際に，HTML の記述が多く見られた．HTML の出力はコンポーネントの更新によって頻繁に変わるため，それに伴ってテストも頻繁に更新する必要があると考えられる．

これらの考察から得られるフロントエンドフレームワークにおけるテストの課題については，以下が考えられる．

- コンポーネントテストは，他の種類のテストに比べ，ソースコードの更新に合わせた修正が必要になる場合が多い．

これらの考察より，開発者に対するフィードバックとしては以下が考えられる．

- コンポーネントテストのアサーションには HTML 出力を用いず，別の方法を用いることで，保守にかかる手間を削減することができる．

6 妥当性への脅威

6.1 内的妥当性

ソフトウェアテストの中には，Flaky Test と呼ばれる，実行するたびに結果が異なる，あるいは実行環境が変化すると結果が異なるテストが存在することが知られている [19]．本研究では，テストの実行結果を単一の環境かつ 1 回の実行のみで評価しているため，実行結果やカバレッジの分析において，そのようなテストの影響を排除できていない可能性がある．

プロジェクトが使用するテストフレームワークによっては，カバレッジを取得することが難しい場合がある．例えば，Mocha, Jasmine, Ava にはカバレッジを取得する機能が標準で備わっていないため，カバレッジを取得するためには外部ツール (c8, nyc など) を使用する必要がある．本調査ではこのような外部ツールを使用したが，カバレッジがうまく取得できなかったため，このようなテストフレームワークを使用しているプロジェクトはカバレッジ取得の対象外とした．また，E2E テストについては，カバレッジを取得すること自体は可能であるが．テストファイルを直接編集する必要があり，負担が大きかったため，本調査では対象外とした．よって，本調査においては，一部のプロジェクトからカバレッジを取得できておらず，カバレッジの分析結果およびカバレッジと他の指標との関係の分析結果には偏りが生じている可能性がある．

6.2 外的妥当性

本研究では調査対象のフロントエンドフレームワークとして，React, Vue.js, Angular, Svelte を選定した．それ以外のフレームワークを調査対象に加えると，本研究における実験結果とは異なる結果が生じる可能性がある．

本研究で収集したプロジェクトは，GitHub のリポジトリ検索 API において，言語を JavaScript, TypeScript, Vue.js, Svelte のいずれかに絞った際のスター数上位 2,500 件のプロジェクトである．以下のような手法によって，収集の対象となるプロジェクトを変更すると，本研究における実験結果とは異なる結果が生じる可能性がある．

- GitHub 以外の場所で公開されているプロジェクトを用いる．
- GitHub におけるリポジトリの検索条件（使用言語・スター数など）を変更する．
- フィルタリングの条件を変更する．

7　関連研究

7.1　オープンソースソフトウェアにおけるテストに関する研究

オープンソースソフトウェアにおけるテストに関する研究が多く存在する．Kochhar らは様々な言語のオープンソースプロジェクトにおけるテストの実施状況や，他の指標との関係について調査した [15]．調査の結果，チームの規模が大きいプロジェクトほどテストケースの数が多いことや，テストケースの数とバグ報告数との相関は弱いことなどがわかった．

Cvitic らは，18 のオープンソースプロジェクトを対象に，テストとその保守状況を調査した [20]．調査の結果，ほとんどのプロジェクトに単体テストと統合テストが存在した一方，ホリスティックテスティングの原則に従っていないことがわかった．

Wang らは，GitHub 上にある 9,129 の機械学習 (DL) プロジェクトにおける単体テストについて調査した [21]．調査の結果，単体テストが存在する DL プロジェクトではプルリクエストの受け入れ率が高いことや，DL プロジェクトの 68% ではテストが全く行われていないことがわかった．

7.2　Web フロントエンドフレームワークに関する研究

近年，Web フロントエンドフレームワークに関する研究が盛んに行われている．Ferreira らは，49 人の開発者を対象に，(1) フロントエンドフレームワークを選択する際に考慮する要因，および (2) フレームワークの移行に関わる計画や労力などについて調査した [22]．(1) の調査の結果，フロントエンドフレームワークを選択する動機となる主な要因は，人気と学習性であることがわかった．また，(2) の調査の結果，調査対象となった開発者の約 1/4 が将来的に別のフレームワークに移行する計画を持っていることがわかった．

Tong らは，複数の Web フロントエンドフレームワークを様々な観点から比較調査した [23]．調査の結果，Next.js が React や Vue.js よりも優れたパフォーマンスを提供することがわかった．

Levlin らは，React，Angular，Vue.js，Svelte の各フレームワークを用いたアプリケーションを対象に，DOM 操作性能やパフォーマンス，コード行数等の比較実験を行った [24]．実験の結果，小規模で軽量なアプリケーションでは Svelte の使用が，大規模アプリケーションでは React や Vue.js の使用が最適であるという結論が得られた．

8 おわりに

本研究では，Web フロントエンドフレームワークを採用した OSS プロジェクトを対象に，テストの導入状況，品質，更新頻度，およびテスト数や品質とバグ報告との関連性を多角的に調査した．その結果，テストの失敗原因のほとんどがフロントエンドフレームワークに特有の実行時エラーであることや，コンポーネント定義ファイルのカバレッジが低いこと，コンポーネントテストはソースコードの変更に合わせた修正が必要であることなど，いくつかの課題が分析できた．また，テストの保守にはフロントエンドフレームワーク特有のテスト失敗理由を知ることが重要であることや，コンポーネントテストのアサーションには HTML 以外を用いるべきであることなど，開発者に対する具体的なフィードバックを提示することができた．

今後の課題としては以下の 2 つが考えられる．

Flaky Test に関する調査　本研究では，Web フロントエンドフレームワークを対象にテストの導入状況や品質について調査したが，テストの信頼性に影響を与える Flaky Test の実態に関する調査は今後の課題として残されている．Flaky Test は，実行環境や繰り返し実行により結果が変わりうる不安定なテストであり，開発者にとって大きな課題となっている．これまでに様々な言語における Flaky Test を調査した論文が複数存在する [19, 25, 26] が，Web フロントエンドフレームワークにおける Flaky Test の実態についてはいまだに明らかにされていない．特に，以下の点について調査が必要である．

- **発生率の特定:** Web フロントエンドフレームワークを使用する OSS プロジェクトにおいてテストを複数回実行することにより，Flaky Test がどの程度の頻度で発生しているかを，定量的に評価する．
- **発生原因の分類:** 非同期処理（API 呼び出しやデータ取得），外部依存（ネットワークや外部サービス），条件付きレンダリングなど，フロントエンド特有の要因が Flaky Test の主な原因であるかを明らかにする．

この調査により，フロントエンドフレームワークを採用したプロジェクトにおいて，Flaky Test を減少させるための具体的なベストプラクティスの提案が可能になると期待される．

テストの修正パターンに関する調査　テストはプロダクションコードの変更に伴って修正されることが多い．本研究では，テストの修正パターンに関する詳細な調査は行わなかったが，テストの保守性や効果的な運用方法を検討するうえで，以下の観点からの調査が今後の重要な課題となる．

- **修正の頻度と要因:** テストコードがどのようなタイミングで，どの程度修正されるかを調査し，頻繁な修正を必要とする箇所の特性を明らかにする．

- **修正内容のパターン化:** 修正されたテストコードを分析し，共通する修正パターンを分類する．例えば，ロケータの変更，Mock データの変更などのパターンを抽出する．

これらの調査により，テストの保守性向上に向けた修正の具体的なガイドラインを提案することが可能となる．

謝辞

本論文の執筆につきましては，多くの方々にご協力をいただきました．

楠本 真二 教授には，研究を行うに当たり，適切な指導をいただきました．特に，中間報告でいただいた多くの質問は，自分では気づかなかった新たな視点を提供してくださり，そのおかげで研究をより良いものにすることができました．

肥後 芳樹 教授には，3 年間，毎週のミーティングから論文の添削に至るまで，担当教員として熱心な指導をしていただきました．特に，研究室を異動された後，多忙な環境であったにも関わらず指導を続けてくださったことについては，感謝してもしきれません．

柗本 真佑 准教授には，研究に関する様々な面で的確なアドバイスをいただきました．特に，中間報告での鋭い指摘のおかげで，研究やプレゼン資料の方向性を大幅に改善できました．

事務員の橋本 美砂子さんには，研究室運営や事務手続きのみならず，出張時のアドバイスや差し入れに至るまで，様々な面でご協力をいただきました．

楠本研究室の先輩方には，私が B4,M1 だった頃に論文の書き方や発表などに関する様々なアドバイスをいただきました．これらのアドバイスのおかげで，論文の執筆をスムーズに行うことができました．

楠本研究室の同期の皆様とは，授業や論文の執筆のみならず，イベント時の買い出しや旅行などを共にすることで，充実した研究室生活を過ごすことができました．

楠本研究室の後輩の皆様には，研究室の業務から日常の何気ない雑談まで，研究室生活のあらゆる面でお世話になりました．

医療従事者の皆様には，修士論文の提出を控えた 1 月にインフルエンザに罹患した際，診察や投薬にご協力いただきました．ご尽力のおかげで無事に治療を終えて元気になり，修士論文の提出を乗り越えることができました．

家族には，決して裕福とはいえない家庭ながらも，約 24 年間，教育や生活など様々な面でご支援をいただきました．就職してからは，今まで支援をいただいた分の恩返しをしたいと思っています．

最後に，本論文の提出までにご協力をいただいたすべての皆様にあらためて感謝を申し上げます．

参考文献

[1] Lazuardy, M. F. S. and Anggraini, D.: Modern front end web architectures with react. js and next. js, *Research Journal of Advanced Engineering and Science*, Vol. 7, No. 1, pp. 132–141 (2022).

[2] Madurapperuma, I., Shafana, M. and Sabani, M.: State-of-art frameworks for Front-end and Back-end Web Development (2022).

[3] Chen, S., Thaduri, U. R. and Ballamudi, V. K. R.: Front-end development in react: an overview, *Engineering International*, Vol. 7, No. 2, pp. 117–126 (2019).

[4] Nelson, B.: Getting to know Vue. js, *Getting to Know Vue. js* (2018).

[5] Tripon, T.-D., Adela Gabor, G. and Valentina Moisi, E.: Angular and Svelte Frameworks: a Comparative Analysis, in *International Conference on Engineering of Modern Electric Systems*, pp. 1–4 (2021).

[6] Bhardwaz, S. and Godha, R.: Svelte.js: The Most Loved Framework Today, in *2023 2nd International Conference for Innovation in Technology*, pp. 1–7 (2023).

[7] Khorikov, V.: *Unit Testing Principles, Practices, and Patterns: Effective Testing Styles, Patterns, and Reliable Automation for Unit Testing, Mocking, and Integration Testing with Examples in C#*, Manning Publications (2021).

[8] IEEE, : ISO/IEC/IEEE International Standard - Systems and Software Engineering – Vocabulary, *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418 (2010).

[9] Tsai, W., Bai, X., Paul, R., Shao, W. and Agarwal, V.: End-to-end integration testing design, in *Annual International Computer Software and Applications Conference*, pp. 166–171 (2001).

[10] Leotta, M., Clerissi, D., Ricca, F. and Tonella, P.: Capture-replay vs. programmable web testing: An empirical assessment during test case evolution, in *2013 20th Working Conference on Reverse Engineering*, pp. 272–281 (2013).

[11] Zhu, H., Hall, P. A. V. and May, J. H. R.: Software unit test coverage and adequacy, *ACM Comput. Surv.*, Vol. 29, No. 4, pp. 366–427 (1997).

[12] Ammann, P. and Offutt, J.: *Introduction to software testing*, Cambridge University Press (2017).

[13] Lakhani, K. R. and Von Hippel, E.: *How open source software works: "free" user-to-user assistance*, Springer (2004).

[14] Dabic, O., Aghajani, E. and Bavota, G.: Sampling Projects in GitHub for MSR Studies, in

2021 IEEE/ACM 18th International Conference on Mining Software Repositories, pp. 560–564 (2021).

[15] Kochhar, P. S., Bissyandé, T. F., Lo, D. and Jiang, L.: An Empirical Study of Adoption of Software Testing in Open Source Projects, in *2013 13th International Conference on Quality Software*, pp. 103–112 (2013).

[16] Wittern, E., Suter, P. and Rajagopalan, S.: A Look at the Dynamics of the JavaScript Package Ecosystem, in *IEEE/ACM Working Conference on Mining Software Repositories*, pp. 351–361 (2016).

[17] Brousse, N.: The issue of monorepo and polyrepo in large enterprises, in *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming*, Programming '19, New York, NY, USA (2019), Association for Computing Machinery.

[18] Kirinuki, H., Tanno, H. and Natsukawa, K.: COLOR: Correct Locator Recommender for Broken Test Scripts using Various Clues in Web Application, in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, pp. 310–320 (2019).

[19] Luo, Q., Hariri, F., Eloussi, L. and Marinov, D.: An empirical analysis of flaky tests, in *22nd ACM SIGSOFT international symposium on foundations of software engineering*, pp. 643–653 (2014).

[20] Cvitic, P. H., Dobslaw, F. and Oliveira Neto, de F. G.: Investigating Software Testing and Maintenance of Open-Source Distributed Ledger, in *International Conference on Software Analysis, Evolution and Reengineering*, pp. 886–896 (2023).

[21] Wang, H., Yu, S., Chen, C., Turhan, B. and Zhu, X.: Beyond Accuracy: An Empirical Study on Unit Testing in Open-source Deep Learning Projects, *ACM Trans. Softw. Eng. Methodol.*, Vol. 33, No. 4 (2024).

[22] Ferreira, F., Borges, H. S. and Valente, M. T.: On the (un-) adoption of JavaScript front-end frameworks, *Software: Practice and Experience*, Vol. 52, No. 4, pp. 947–966 (2022).

[23] Tong, J., Jikson, R. R. and Gunawan, A. A. S.: Comparative Performance Analysis of Javascript Frontend Web Frameworks, in *International Conference on Electronic and Electrical Engineering and Intelligent System*, pp. 81–86 (2023).

[24] Levlin, M., Soini, A. and Truscan, D.: DOM benchmark comparison of the front-end JavaScript frameworks React, Angular, Vue, and Svelte (2020).

[25] Gruber, M., Lukasczyk, S., Kroiß, F. and Fraser, G.: An empirical study of flaky tests in python, in *IEEE Conference on Software Testing, Verification and Validation*, pp. 148–158

(2021).

[26] Hashemi, N., Tahir, A. and Rasheed, S.: An empirical study of flaky tests in javascript, in *IEEE International Conference on Software Maintenance and Evolution*, pp. 24–34 (2022).