올바른 테스트 작성을 위한 규칙

실무에 바로 적용하는 프런트엔드 테스트

이재성

올바른 테스트 작성을 위한 규칙은 뭐가 있을까?

인터페이스를 기준으로 테스트를 작성하자

인터페이스를 기준으로 테스트를 작성하자

서로 다른 클래스 또는 모듈이 상호작용하는 시스템

인터페이스를 기준으로 테스트를 작성하자

서로 다른 클래스 또는 모듈이 상호작용하는 시스템

내부 구현에 대한 테스트는 캡슐화를 위반

깨지기 쉬운 테스트

인터페이스를 기준으로 테스트를 작성하자

잘못된 테스트 코드

```
// 👨‍✈️
it('isShowModal 상태를 true로 변경했을 때 ModalComponent의 display 스타일이 block이며,
"안녕하세요!" 텍스트가 노출된다.', () => {
  // 구현에 종속적인 코드와 복잡한 상태 변경 코드들이 발생할 수 있습니다.
  SpecificComponent.setState({ isShowModal: true });

  // ...
});
```

인터페이스를 기준으로 테스트를 작성하자

잘못된 테스트 코드

```
// 👨‍✈️
it('isShowModal 상태를 true로 변경했을 때 ModalComponent의 display 스타일이 block이며,
"안녕하세요!" 텍스트가 노출된다.', () => {
  // 구현에 종속적인 코드와 복잡한 상태 변경 코드들이 발생할 수 있습니다.
  SpecificComponent.setState({ isShowModal: true });

  // ...
});
```

1. 변경되는 상태가 많은 경우 테스트 코드 상에서
일일이 직접 변경해야 하며
어떤 상황에서 변경 되는 것인지 드러나지 않음

인터페이스를 기준으로 테스트를 작성하자

잘못된 테스트 코드

```
// 👨‍🔧
it('isShowModal 상태를 true로 변경했을 때 ModalComponent의 display 스타일이 block이며,
"안녕하세요!" 텍스트가 노출된다.', () => {
  // 구현에 종속적인 코드와 복잡한 상태 변경 코드들이 발생할 수 있습니다.
  SpecificComponent.setState({ isShowModal: true });

  // ...
});
```

2. 내부 상태나 변수값을 기준으로 검증하다 보니
어떤 것을 검증하는지 테스트 코드만 보고 한 눈에 파악하기 어려움

인터페이스를 기준으로 테스트를 작성하자

잘못된 테스트 코드

```
// 👮
it('isShowModal 상태를 true로 변경했을 때 ModalComponent의 display 스타일이 block이며,
"안녕하세요!" 텍스트가 노출된다.', () => {
  // 구현에 종속적인 코드와 복잡한 상태 변경 코드들이 발생할 수 있습니다.
  SpecificComponent.setState({ isShowModal: true });

  // ...
});
```

3. 내부 구현을 검증하려다 보니 구현에 종속적인 테스트 코드가 양산됨
상태나 변수명이 하나라도 바뀌면 테스트 코드 모두를 바꿔야 함
즉, 캡슐화를 위반하는 코드가 된다.

인터페이스를 기준으로 테스트를 작성하자

올바른 테스트 코드

```
// 👩‍🦱
it('버튼을 누르면 모달을 띄운다.', () => {
  // 유저의 동작과 비슷하도록 클릭 이벤트를 발생
  user.click(screen.getByRole('button'));

  //...
});
```

✔ 내부 구현과 종속성이 없으며 캡슐화에 위반되지 않음

✔ 어떤 행위를 하는지 명확해짐

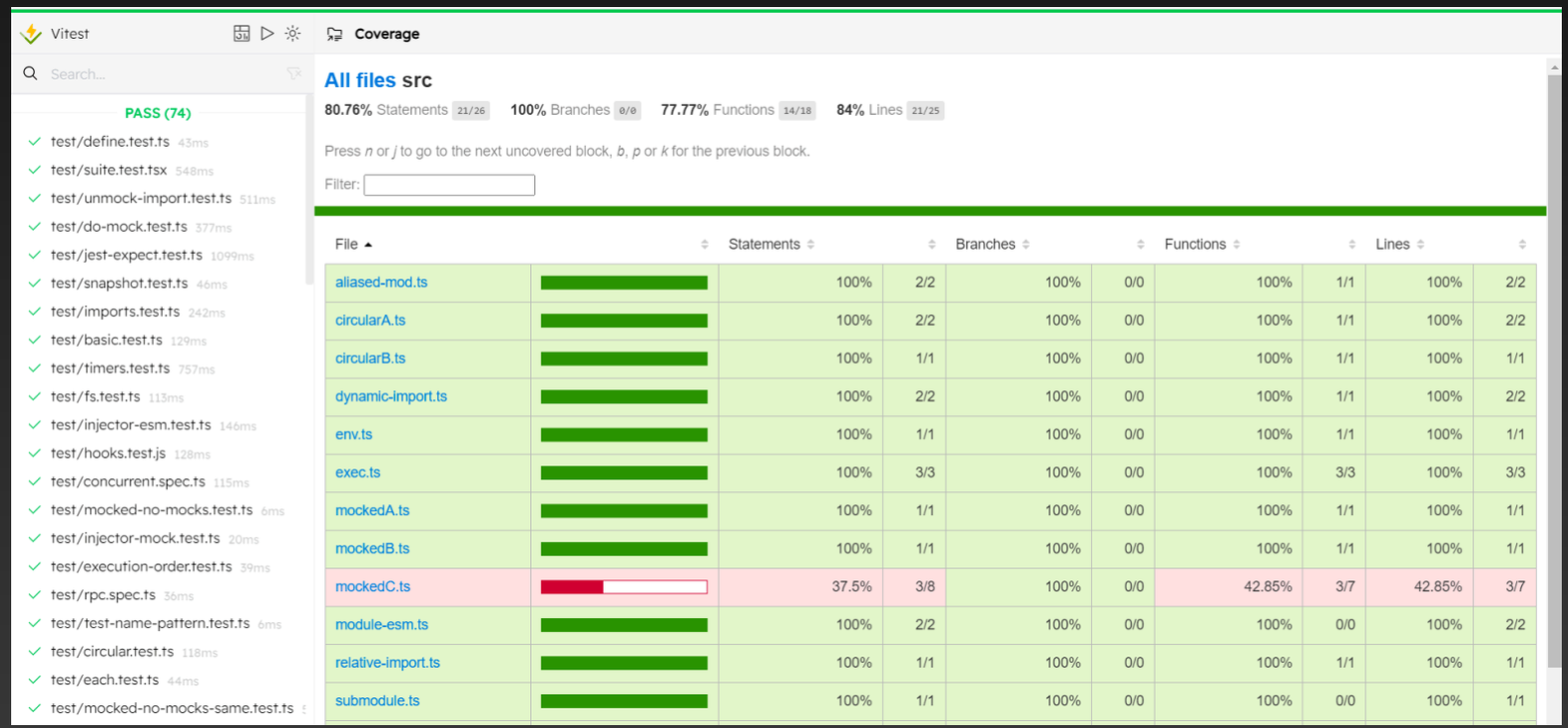
✔ 테스트를 설명하기 위한 불필요한 주석이나 설명 또한 없음

커버리지 보다는 의미있는 테스트인지 고민하자

커버리지 보다는 의미있는 테스트인지 고민하자

✔ 테스트 코드가 프로덕션 코드의 몇 %를 검증하고 있는지 나타낸 지표

✔ 구문(statements), 분기(Branches), 함수(Functions), 줄(Lines) 등을 기준으로 함



예) vitest 테스트 커버리지 결과

100%

그럼 커버리지 100%를 목표로 하는게 맞을까?

100% 테스트 커버리지에 의존하는게 맞을까?

✔ 테스트 작성, 실행, 유지 보수 측면에서 너무 많은 비용이 발생

✔ 100% 커버리지로 테스트를 작성했어도 잘못된 검증때문에 문제가 발생할 수 있음

함수의 모든 부분을 커버

하지만 아무것도 검증하고 있지 않음

```
function isLargerThan5(value) {
  if (typeof value !== 'number') {
    return false;
  }

  return value > 5;
}

it('isLargerThan5 test', () => {
  const result = isLargerThan5(100);
  const result2 = isLargerThan5('hello');
});
```

커버리지 보다는 의미있는 테스트인지 고민하자

테스트가 필요할까? 🤔

```jsx
import React from 'react';

const List = ({ items = [] }) => {
  return (
    <ul>
      {items.map((data) => {
        return (
          <li key={data}>{data}</li>
        );
      })}
    </ul>
  );
};

export default List;
```

단순한 UI 렌더링

커버리지 보다는 의미있는 테스트인지 고민하자

테스트가 필요할까? 🤔

```
export const isNumber = (value) => typeof value === 'number';
export const isString = (value) => typeof value === 'string'
```

커버리지 보다는 의미있는 테스트인지 고민하자

커버리지만을 위한 검증은
의미없는 테스트가 될 가능성이 높다

100% 커버리지를 위한 테스트 보다는..

✅ 의미 있는 테스트인지?

✅ 어떤 범위까지 검증해야 효율적인 테스트가 될지?

고민해야 한다

테스트 코드도 유지 보수의 대상!
가독성을 높이자

테스트 코드도 유지 보수의 대상!
가독성을 높이자

1. 테스트 하고자 하는 내용을 명확하게 적자

테스트 코드도 유지 보수의 대상!
가독성을 높이자

1. 테스트 하고자 하는 내용을 명확하게 적자

검증 기능: 리스트에서 체크된 항목들을 삭제

```
// 🙅‍♂️
it('리스트에서 항목이 제대로 삭제된다.', () => {
  // ...
})

// 🙆‍♀️
it('항목들을 체크한 후 삭제 버튼을 누르면 리스트에서 체크된 항목들이 삭제된다.', () => {
  // ...
})
```

테스트 코드도 유지 보수의 대상!
가독성을 높이자

2. 하나의 테스트에서는 가급적 하나의 동작만 검증하자

테스트 코드도 유지 보수의 대상!
가독성을 높이자

2. 하나의 테스트에서는 가급적 하나의 동작만 검증하자

단일 책임 원칙(SRP, Single Responsibility Principle)

모든 클래스는 하나의 책임을 갖고 그와 관련된 책임을 캡슐화하여
변경에 견고한 코드를 만들어야 한다

테스트 코드도 유지 보수의 대상!
가독성을 높이자

2. 하나의 테스트에서는 가급적 하나의 동작만 검증하자

단일 책임 원칙(SRP, Single Responsibility Principle)

모든 클래스는 하나의 책임을 갖고 그와 관련된 책임을 캡슐화하여
변경에 견고한 코드를 만들어야 한다

테스트에서도 동일!

테스트 코드도 유지 보수의 대상!
가독성을 높이자

2. 하나의 테스트에서는 가급적 하나의 동작만 검증하자

```
// 👮
it('장바구니에 담긴 상품들이 정상적으로 노출되고, 수량을 변경하면 가격이 재계산된다. 그리고
삭제 버튼을 누르면 상품이 삭제된다.', () => {
  // ...
})
```

테스트 코드도 유지 보수의 대상!
가독성을 높이자

2. 하나의 테스트에서는 가급적 하나의 동작만 검증하자

```
// 👮
✔ it('장바구니에 담긴 상품들을 정상적으로 렌더링 한다.', () => {
    // ...
})

✔ it('장바구니에 담긴 상품의 수량을 수정하면 가격이 재계산된다.', () => {
    //...
})

✔ it('장바구니에 담긴 항목의 삭제 버튼을 누르면 리스트에서 삭제된다.', () => {
    //...
})
```

정리

올바른 테스트 작성을 위한 규칙으로는,

인터페이스를 기준으로 테스트를 작성하자

- 내부 구현에 대한 테스트 코드는 강한 의존성 때문에 깨지기 쉽고 유지보수 하기 어렵다.
- 인터페이스를 기준으로 캡슐화에 위반되지 않으며 종속성이 없는 테스트를 작성하자

100% 커버리지보다는 의미 있는 테스트인지 고민하자

- 커버리지를 쫓다 보면 큰 유지 보수 비용이 발생하며 제대로 된 검증이 되었다는 착각이 들 수 있다.
- 의미 있는 테스트가 무엇인지 고민해보자!

테스트 코드도 유지 보수 의 대상이다. 가독성을 높이자!

- 테스트 하기 전 테스트 하고자 하는 내용을 명확하게 적자
- 하나의 테스트에는 가급적 하나의 동작만 검증하자