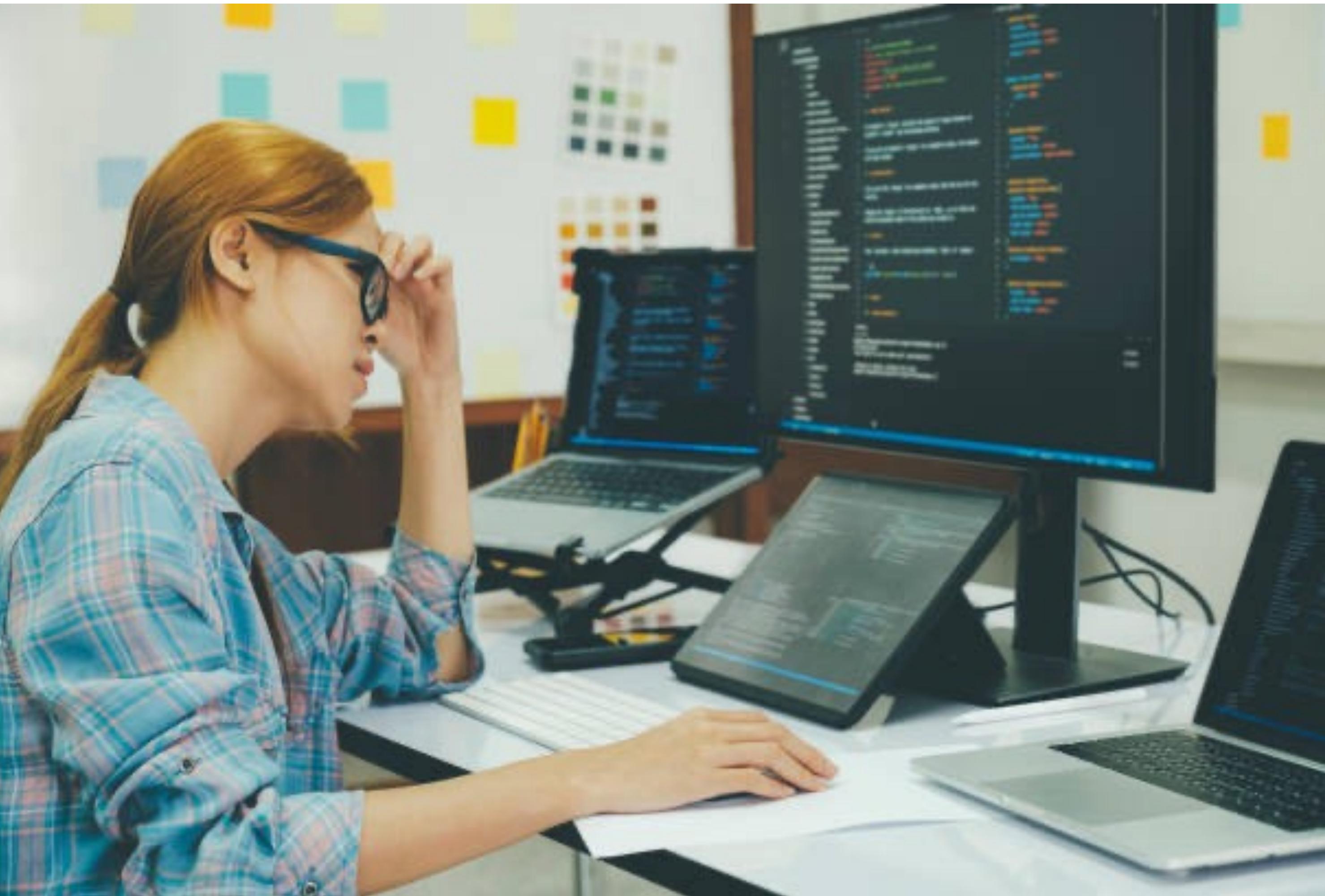


4 테스트 구축하기

2023 02 19 정요한

자가 테스트 코드의 가치



자가 테스트 코드의 가치

- 프로그래머라면 실제 코드를 작성하는 시간의 비중은 그리 크지 않다. 현재 상황을 파악하기도 하고, 설계를 고민하기도 한다. 대부분의 시간은 디버깅에 쓴다. 디버깅하느라 밤늦게까지 고생한 경험이 있을 것이다. 디버깅 무용담을 하나씩 간직하고 있다. 버그 수정 자체는 금방 끝난다. 진짜 끔찍한 건 버그를 찾는 여정이다. 또한 **버그를 잡는 과정에서 다른 버그를 심기도 하는데**, 그 사실을 한참이 지나서야 알아채기도 한다. 그리고 **또 다시 버그를 찾느라 수많은 시간을 날린다.**
- 컨퍼런스서, **클래스마다 테스트 코드를 갖춰야한다.**는 말을 듣고, 프로덕션 코드와 테스트 코드를 함께 담기로 결정했다. 당시에 Iterative Programming을 따르고 있었기 때문에 반복 주기가 하나 끝날 때마다 가능하면 테스트 코드도 추가했다. 반복 주기는 1주 정도였는데, 테스트 작업은 꽤 쉽고 간단했지만 상당히 지루했다. 테스트 코드가 콘솔에 출력한 결과를 일일이 눈으로 확인해야 했기 때문이다. 난 상당히 게을러서 일을 줄이는 작업이라면 얼마든지 감수할 각오가 돼 있다. 테스트가 성공했는지 확인하려면 의도한 결과와 테스트 결과가 같은지만 비교하면 된다. 그래서 모든 테스트가 성공화면에 OK만 출력하도록 만들었다.
- **모든 테스트를 완전히 자동화하고 그 결과까지 스스로 검사하게 만들자.** 컴파일 할 때마다 테스트도 함께 했고, 곧바로 생산성이 급상승했다. 디버깅 시간이 크게 줄어든 것이다. 가장 최근 테스트로 잡은 버그를 다시 살펴보면 테스트에 걸려 눈에 확 드러난다. 직전까지 테스트가 성공했다면 마지막 테스트 이후에 작성한 코드에서 버그가 발생했음을 알 수 있다.
- 반복주기를 굳이 기다리지 않고 함수 몇 개만 작성해도 곧바로 테스트를 추가. 매일 두어 개의 새로운 기능과 그에 딸린 테스트 코드가 쌓여갔다. 그래서 **regression bug**를 잡는데 몇 분 이상 걸린 적이 거의 없다. **Test Suite는 강력한 버그 검출 도구로, 버그를 찾는 데 걸리는 시간을 대폭 줄여준다.**
- **테스트를 작성하기 가장 좋은 시점은 프로그래밍을 시작하기 전이다.** 나는 기능을 추가해야 할 때 테스트부터 작성한다. 테스트를 작성하다 보면 원하는 기능을 추가하기 위해 무엇이 필요한지 고민하게 된다. 구현보다 인터페이스에 집중하게 된다는 장점도 있고, **코딩이 완료되는 시점을 정확하게 판단할 수 있다.** 테스트를 모두 통과한 시점이 바로 코드를 완성한 시점이다. 켄트백은 테스트부터 작성하는 습관을 바탕으로 TDD란 기법을 창시했다. 테스트를 작성하고, 이 테스트를 통과하게끔 코드를 작성하고, 결과 코드를 최대한 깔끔하게 리팩터링하는 과정을 짧은 주기로 반복한다. 이러한 테스트-코딩-리팩터링 과정을 한 시간에도 여러차례 진행하기 때문에 코드를 대단히 생산적이면서도 차분하게 작성할 수 있다.
- **리팩터링하고 싶다면 테스트를 반드시 작성해야 한다.**

테스트할 샘플 코드

```
import {Province} from './Province'; export interface IProducer {
  import { IProducer } from './types';
  name: string;
  cost: number;
  production: number;
}

export class Producer {
  private _province;
  private _cost : number;
  private _name: string;
  private _production: number;

  constructor(aProvince: Province, data: IProducer) {
    this._province = aProvince;
    this._cost = data.cost;
    this._name = data.name;
    this._production = data.production || 0;
  }

  get name() {
    return this._name;
  }

  get cost():number {
    return this._cost;
  }

  set cost(arg: number) {
    this._cost = arg;
  }

  get production():number {
    return this._production;
  }

  set production(amount: number) {
    const newProduction = Number.isNaN(amount) ? 0 : amount;
    this._province.totalProduction += newProduction -
    this._production;
    this._production = newProduction;
  }
}
```

```
import {Producer} from './Producer';
import { IProducer } from './types';

export class Province {
  private _name: string;
  private _producers: IProducer[];
  private _totalProduction: number;
  private _demand: number;
  private _price: number;

  constructor(doc: Province) {
    this._name = doc.name;
    this._producers = [];
    this._totalProduction = 0;
    this._demand = doc.demand;
    this._price = doc.price;
    doc.producers.forEach((d) => this.addProducer(new Producer(this, d)));
  }

  get name(): string {
    return this._name;
  }

  get producers(): IProducer[] {
    return this._producers.slice();
  }

  get totalProduction() {
    return this._totalProduction;
  }

  set totalProduction(arg) {
    this._totalProduction = arg;
  }

  get demand(): number {
    return this._demand;
  }

  set demand(arg: number) {
    if (arg < 0 || Number.isNaN(` ${arg}`)) {
      this._demand = Number.NaN;
    } else this._demand = arg;
  }

  get price(): number {
    return this._price;
  }

  set price(arg: number) {
    this._price = arg;
  }

  addProducer = (arg: Producer) => {
    this._producers.push(arg);
    this._totalProduction += arg.production;
  };

  get shortfall() {
    return this._demand - this.totalProduction;
  }

  get profit() {
    return this.demandValue - this.demandCost;
  }

  get demandValue() {
    return this.satisfiedDemand * this.price;
  }

  get satisfiedDemand() {
    return Math.min(this._demand, this.totalProduction);
  }

  get demandCost() {
    let remainingDemand = this.demand;
    let result = 0;
    this._producers
      .sort((a, b) => a.cost - b.cost)
      .forEach((p) => {
        const contribution = Math.min(remainingDemand, p.production);
        remainingDemand -= contribution;
        result += contribution * p.cost;
      });
    return result;
  }
}
```

첫 번째 테스트

- 코드를 테스트하기 위해선 먼저 테스트 프레임워크를 설치해야한다. Mocha(<https://github.com/mochajs/mocha>), jest(<https://github.com/facebook/jest>) 등이 있다.
- describe(TestSuite?)/it(test case) 에 string 값에 테스트에 대한 설명을 넣는 분도 있고, 함수 본문에만 충실하게 하고 string 값을 비워두시는 분도 있다. 저자는 실패한 테스트가 뭔지 식별할 수 있는 정도로 작성하는 편이다.
- **실패해야 할 상황에서는 반드시 실패하게 만들자.** 수많은 테스트를 실행했음에도 실패하는게 없다면 테스트가 내 의도와는 다른 방식으로 코드를 다루는 건 아닌지 불안해진다. 그래서 각각의 테스트가 실패하는 모습을 최소한 한 번씩은 직접 확인해본다. 일시적으로 코드에 오류를 주입하는 것이다.
- **자주 테스트하라.** 작성 중인 코드는 최소한 몇 분 간격으로 테스트하고, 적어도 하루에 한 번은 전체 테스트를 돌려보자.
- 실전에서는 테스트케이스의 수는 수천 개 이상일 수도 있다. 뛰어난 테스트 프레임워크를 사용한다면 이렇게 많은 테스트도 간편하게 실행할 수 있고 무언가 실패한다면 금방 확인할 수 있다. **간결한 피드백은 자가 테스트에서 매우 중요하다.** 테스트를 굉장히 자주한다. 방금 추가한 코드에 문제가 없는지 혹은 리팩터링하면서 실수한 것은 없는지 확인하기 위해서다.
- 테스트 환경은 다르다. 일반적으로 GUI 환경에서는 빨간/초록으로 표시된다. **실패한 테스트가 하나라도 있으면 리팩터링 하면 안 된다.** => “빨간 막대일 때는 리팩터링하지 말라” 최근 변경을 취소하고 마지막으로 모든 테스트를 통과했던 상태로 되돌아가라 => “초록 막대로 되돌려라”. 버전관리 소프트웨어로 체크포인트로 돌아간다.
- **GUI 또는 콘솔이든, 간단하게 키 하나만 누르면 모든 테스트가 실행되도록 하고, 모든 테스트가 통과했다는 사실을 빨리 알도록 하자.**

테스트할 샘플 코드 2 (fixture / factory method)

```
export const SampleProvinceData = () : object => {
  return {
    name: "Asia",
    producers: [
      { name: "Byzantium", cost: 10, production: 9},
      { name: "Attalia", cost: 12, production: 10},
      { name: "Sinope", cost: 10, production: 6}
    ],
    demand: 30,
    price: 20
  }
}
```

테스트 추가하기

- public 메서드를 빠짐없이 테스트하는 방식이 아닌, **테스트는 위험 요인을 중심으로 작성해야 한다.** 테스트의 목적은 어디까지나 현재 혹은 향후에 발생하는 버그를 찾는데 있다. 따라서 단순히 필드를 읽고 쓰기만 하는 접근자는 테스트를 할 필요가 없다. 이런 코드는 너무 단순해서 버그가 숨어들 가능성도 별로 없다. 그런데 말이죠. 여기 챕터의 예에서는, **string => number**로 변환하는 **setter** 정도, 연산에 의해서 결정되는 **getter**는 테스트 할 수준이다.
- **테스트를 너무 많이 만들다 보면 오히려 필요한 테스트를 놓치기 쉽기 때문에 아주 중요한 포인트다.** 적은 수의 테스트만으로 큰 효과를 얻고 있다. 잘못될까봐 가장 걱정되는 영역을 집중적으로 테스트하는데, 이렇게 해서 테스트에 쓸는 노력의 효과를 극대화 하는 것이다.
- **완벽하게 만드느라 테스트를 수행하지 못하느니, 불완전한 테스트라도 작성해 실행하는 게 낫다.** 격언 중에, **done is better than perfect.** 겁먹지 말고 우선 작성하라.
- 똑같은 픽스처를 재활용하더라도 테스트끼리 상호작용하지 않도록 유의하자. 다른 테스트가 실패할 수가 있다. 테스트를 실행하는 순서에 따라 결과가 달라질 수 있다.
- 결합도에 유의. const 여도 방심하지마라. 객체의 참조가 상수이므로 객체의 ‘내용’이 변경될 수 있다. 참고 <https://jojoldu.tistory.com/611>

```
describe('province with fixture', ()=> {
  const asia: Province = new Province(SampleProvinceData() as Province);
```



```
describe('province with fixture', ()=> {
  let asia: Province;
  beforeEach(()=>{
    asia = new Province(SampleProvinceData() as Province)
  })

  it('shortfall', () => {
    expect(asia.shortfall).toEqual(5);
  })

  it('profit', ()=>{
    expect(asia.profit).toEqual(230);
  })

  it('change production', ()=>{
    asia.producers[0].production = 20;
    expect(asia.shortfall).toEqual(-6);
    expect(asia.profit).toEqual(292);
  })
})
```

fixture 수정하기

- fixture를 불러와서 값을 수정한 뒤 테스트를 하는 경우도 있다. 수정하는 세터 부분은 단순하여 버그가 생길 일은 별로 없지만, 저서의 경우 복잡한 동작을 하기에 테스트를 할 수 있다. fixture를 사용하여 setup을 하고, 테스트를 수행하는 과정을 설정-실행-검증(setup-exercise-verify)가 있다.
- 초기 준비 작업 중 공통되는 부분을 mocha, jest에서는 beforeEach 와 같은 표준 설정 루틴에 모아 처리하기도 한다.
- <https://jojoldu.tistory.com/611> 팩토리 메서드 활용도 고려해보자.
- teardown 부분은 테스트 마무리 단계에서 픽스처를 클린업하는 단계. teardown처럼 테스트 프레임워크에서 픽스처를 알아서 깔끔하게하는 기능도 있다.

경계 조건 검사하기

- 작성한 테스트는 모든 일이 순조롭고 사용자도 우리 의도대로 사용하는, 꽃길happy path 상황. 해피 패스를 벗어나면 어떤 일이 벌어지는지 확인하는 테스트도 함께 작성하면 좋다. 숫자형이라면 0일 때, 음수일 때, 문자열을 넣어보자. 생성자에 불가능한 값을 넣고 의도대로 생성되는지 확인하자. 이런 경우는 리팩터링 하기 전에 작성되어 있어야한다. 리팩터링은 겉보기 동작에는 영향을 주지 않아야하며, 이런 오류는 겉보기 동작에 해당하지 않는다. 경계조건에 대응하는 동작이 리팩터링 때문에 변하는지는 신경 쓸 필요 없다.
- 문제가 생길 가능성이 있는 경계 조건을 생각해보고 그 부분을 집중적으로 테스트하자
- 어차피 모든 버그를 잡아낼 수는 없다고 생각하여 테스트를 작성하지 않는다면 대다수의 버그를 잡을 수 있는 기회를 날리는 셈이다 **done is better than perfect**.
- 반대로, 수확 체감 법칙. 테스트를 너무 많이 작성하다보면 오히려 의욕이 떨어 질 수 있으니, 위험한 부분에 집중해보자. 처리과정이 복잡하거나, 오류가 생길만한 부분을 찾아보자.
- 테스트가 모든 버그를 걸러주진 못하더라도, 안심하고 리팩터링할 수 있는 보호막은 되어준다. 그리고 리팩터링을 하면서 프로그램을 더욱 깊이 이해하게되어 더 많은 버그를 찾게 된다. 나는 항상 테스트 스위트부터 갖춘 뒤에 리팩터링하지만, 리팩터링하는 동안에도 계속해서 테스트를 추가한다.

끝나지 않은 여정

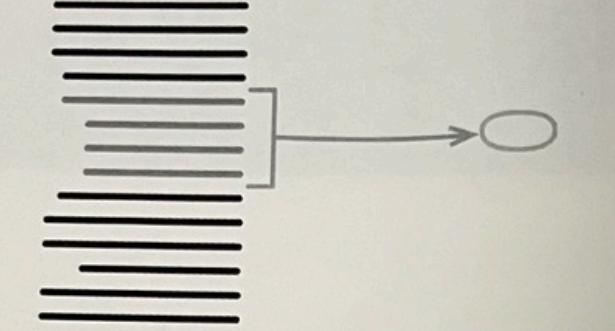
- 다시 한번 강조. 리팩터링을 하려면 테스트 코드가 반드시 필요하다.
- 단위 테스트는 코드의 작은 영역만을 대상으로 빠르게 실행되도록 설계된 테스트. 자가 테스트 시스템 대부분 단위 테스트가 차지한다. 단위 테스트는 자가 테스트 코드의 핵심.
- 물론 컴포넌트 사이의 상호작용에 집중하는 테스트나, 소프트웨어의 다양한 계층의 연동을 검사하는 테스트, 성능 문제를 다루는 테스트 등 다양한 유형의 테스트가 있다.
- 한 번에 완벽한 테스트를 갖출 순 없다. 제품 코드에 못지 않게 테스트 스위트도 지속해서 보강한다. 다시 말해 기능을 새로 추가할 때마다 테스트도 추가하는 것은 물론, 기존 테스트도 다시 살펴본다. 기존 테스트가 충분히 명확한지, 테스트 과정을 더 이해하기 쉽게 리팩터링할 수는 없는지, 제대로 검사하는지 등을 확인한다.
- **버그 리포트(이슈)를 받으면 가장 먼저 그 버그를 드러내는 단위 테스트 부터 작성하자**
- **어느 정도 하면 충분히 테스트했다고 할 수 있나요?**라는 질문에 정확한 기준은 없다. 테스트 커버리지를 기준으로 삼는 사람도 있지만, 테스트 커버리지는 코드에서 테스트하지 않은 영역을 찾는 데만 도움될 뿐, 테스트 스위트의 품질과는 크게 상관 없다. 테스트 스위트가 충분한지를 평가하는 기준은 주관적이다. 누군가 결함을 심으면 테스트가 발견할 수 있다는 믿음을 기준으로 할 수 있다. 개관적으로 측정할 수 없어서 헛된 믿음 뿐인지 알 도리가 없다. 하지만, **테스트 코드의 목적은 이 믿음을 갖게 해주는 것이다.** 테스트 결과가 초록색인 것만 보고도 리팩터링 과정에서 생겨난 버그가 하나도 없다고 확신할 수 있다면, 충분히 좋은 테스트 스위트라 할 수 있다.
- **테스트를 너무 많이 작성할 가능성도 있다. (수확체감 법칙)** 제품 코드보다 테스트 코드를 수정하는 데 시간이 더 걸린다면, 그리고 테스트 때문에 개발 속도가 느려진다고 생각되면 테스트를 과하게 작성한건 아닌지 의심해보자. 그런데 말이죠. **너무 많은 경우보다는 너무 적은 경우가 훨씬 훨씬 많다.**

5 리팩터링 카탈로그 보는 법

리팩터링 설명 형식

- 6장부터 12장 까지 이 규칙을 따릅니다.
- 이름 : 가장 먼저 나오는 항목. 같은 기법을 다르게 부르는 경우도 있기 때문에 흔하게 사용되는 이름도 함께 소개. 이전 버전에서 사용된 이름도 함께 표현합니다. 반대 리팩터링 기법의 이름도 함께 소개.
- 개요 : 핵심 개념을 간략히 표현한 개념도와 코드 예시.

6.1 함수 추출하기
Extract Function



- 반대 리팩터링: 함수 인라인하기^{6.2절}
- 1판에서의 이름: 메서드 추출

```
function printOwing(invoice) {  
    printBanner();  
    let outstanding = calculateOutstanding();  
  
    //세부 사항 출력  
    console.log(`고객명: ${invoice.customer}`);  
    console.log(`채무액: ${outstanding}`);  
  
    function printDetails(outstanding) {  
        console.log(`고객명: ${invoice.customer}`);  
        console.log(`채무액: ${outstanding}`);  
    }  
}
```

리팩터링 설명 형식

- 배경 : 리팩터링 기법이 왜 필요한지와 그 기법을 적용하면 안 되는 상황을 설명. 썰도 있음.
- 절차 : 리팩터링 하는 과정. 구체적인 단계를 잊지 않도록 개인 노트에 기록해 둔 것이라 압축 된 표현이 많다. 절차에는 해당 리팩터링의 단계를 가능한 한 잘게 나눠놓았다. 리팩터링을 안전하게 수행하려면 단계를 최대한 잘게 나누고 각 단계마다 테스트해야 한다. 실전에서는 각 단계를 더 크게 잡고 리팩터링 하고, 버그가 생기면 이전 단계로 돌아가서 크기를 줄여 다시 진행한다. ‘->’로 시작하는 세부 항목이 딸려 있는데, 세부 단계가 아니라 특수한 경우를 위한 참고사항이다. 체크리스트처럼 활용할 수 있으며, 나도 여기 적어둔건 잘 기억 못 한다.

절차에서는 한 가지 방식만 소개할 때도 있는데, 해당 리팩터링을 수행하는 유일한 방법이라는 뜻은 아니다. 그저 대부분 상황에 잘 들어맞는 절차를 선정했을 뿐이다. 구체적인 절차를 변형하게 될텐데, 그래도 문제 없다. 작은 단계를 밟아 나가는게 핵심이다. 상황이 난해할 수록 단계를 만들자.
- 예시 : 리팩터링 기법을 실제로 적용하는 간단한 예와 효과. 간단한 사례로 구성했다.

주의를 분산시키지 않고 리팩터링의 기본에 집중하여 설명하기 위해서다. 예시는 어디까지나 해당 절에서 소개하는 리팩터링 기법 하나만 보여주기 위한 것이다. 그래서 수정을 마친 코드에도 다른 문제가 얼마든지 남아 있을 수 있다. 이를 해결하려면 다른 리팩터링을 적용해야한다.

간혹 실무에서 하나의 패턴처럼 연달아 적용되는 리팩터링들이 있는데, 이런 경우에는 예시에서도 연달아 적용했다. 그외 대부분의 경우는 한 가지 리팩터링만 적용한 상태로 됬다. 리팩터링 카탈로그의 주 용도는 어디까지나 레퍼런스이기 때문에 각각의 리팩터링을 최대한 독립적으로 구성하기 위해서다.

신규 코드 덩어리를 추출한 경우
당 코드 덩어리를 추출한 경우

절차

① 함수를 새로 만들고 목적을 잘 드러내는 이름을 붙인다(‘어떻게’가 아닌 ‘무엇을’ 하는지가 드러나야 한다).
→ 대상 코드가 함수 호출문 하나처럼 매우 간단하더라도 함수로 뽑아서 목적이 더 잘 드러나는 이름을 붙일 수 있다면 추출한다. 이런 이름이 떠오르지 않는다면 함수로 추출하면 안 된다는 신호다. 하지만 추출하는 과정에서 좋은 이름이 떠오를 수도 있으니 처음부터 최선의 이름부터 짓고 시작할 필요는 없다. 일단 함수로 추출해서 사용해보고 효과가 크지 않으면 다시 원래 상태로 인라인해도 된다. 그 과정에서 조금이라도 깨달은 게 있다면 시간 낭비는 아니다. 중첩 함수를 지원하는 언어를 사용한다면 추출한 함수를 원래 함수 안에 중첩시킨다. 그러면 다음 단계에서 수행할 유효범위를 벗어난 변수를 처리하는 작업을 줄일 수 있다. 원래 함수의 바깥으로 꺼내야 할 때가 오면 언제든 함수 옮기기^{8.1절}를 적용하면 된다.

- ② 추출할 코드를 원본 함수에서 복사하여 새 함수에 붙여넣는다.
- ③ 추출한 코드 중 원본 함수의 지역 변수를 참조하거나 추출한 함수의 유효범위를 벗어나는 변수는 없는지 검사한다. 있다면 매개변수로 전달한다.
- 원본 함수의 중첩 함수로 추출할 때는 이런 문제가 생기지 않는다.
- 일반적으로 함수에는 지역 변수와 매개변수가 있기 마련이다. 가장 일반적인 처리 방법은 이런 번호는 고드에 방금 추출한 것과 똑같거나 비슷한 코드가 없는지 살핀다. 있다면 방금 추출한 새 함수를 호출하도록 바꿀지 검토한다(인라인 코드를 함수 호출로 바꾸기^{8.5절}).
- 중복 혹은 비슷한 코드를 찾아주는 리팩터링 도구도 있다. 이런 도구가 없다면 검색 기능을 이용하여 다른 곳에 중복된 코드가 없는지 확인해보는 것이 좋다.

예시: 유효범위를 벗어나는 변수가 없을 때

아주 간단한 코드에서는 함수 추출하기가 굉장히 쉽다.

```
function printOwing(invoice) {  
    let outstanding = 0;  
  
    console.log("*****");  
    console.log("**** 고객 채무 ****");  
    console.log("*****");  
  
    // 미해결 채무(outstanding)를 계산한다.  
    for (const o of invoice.orders) {  
        outstanding += o.amount;  
    }  
  
    // 마감일(dueDate)을 기록한다.  
    const today = Clock.today;  
    invoice.dueDate = new Date(today.getFullYear(), today.getMonth(),  
        today.getDate() + 30);  
  
    // 세부 사항을 출력한다.  
    console.log(`고객명: ${invoice.customer}`);  
    console.log(`채무액: ${outstanding}`);  
    console.log(`마감일: ${invoice.dueDate.toLocaleDateString()}`);
```

6 기본적인 리팩터링

6.1 함수 추출하기

개요 AS IS

```
const printOwing = (invoice) => {
  let outstanding = 0;

  console.log("*****");
  console.log("**** 고객 채무 ****");
  console.log("*****");

  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  const today = clock.today;
  invoice.dueDate = new Date(today.getFullYear(),
                            today.getMonth(),
                            today.getDate());

  console.log(`고객명: ${invoice.customer}`);
  console.log(`채무액: ${outstanding}`);
  console.log(`마감일: ${invoice.dueDate.toLocaleDateString()}`);
}
```

TO BE

```
const printOwing = (invoice) => {
  printBanner();
  printDetails(
    recordDueDate(invoice),
    calculateOutStanding(invoice)
  );
}

const printBanner =() => {
  console.log("*****");
  console.log("**** 고객 채무 ****");
  console.log("*****");
}

const printDetails =(invoice, outstanding) => {
  console.log(`고객명: ${invoice.customer}`);
  console.log(`채무액: ${outstanding}`);
  console.log(`마감일: ${invoice.dueDate.toLocaleDateString()}`);
}

const recordDueDate = (invoice) => {
  const result = { ... invoice }
  result.dueDate = new Date(clock.today.getFullYear(),
                            clock.today.getMonth(),
                            clock.today.getDate())

  return result
}

const calculateOutStanding = (invoice) => {
  return invoice.orders.reduce((result, order) => order.amount + result, 0)
}
```

반대 리팩터링: 함수 인라인하기

함수 추출하기 (배경)

- 가장 많이 사용하는 리팩터링 중 하나다. 함수라고 표현했지만 method/procedure/subroutine 에도 똑같이 적용된다.
- 코드 조각을 찾아 무슨일을 하는지 파악한 다음, **독립된 함수로 추출하고 목적에 맞는 이름을 붙인다.**
- 언제 독립적으로 뭉나? 길이 기준으로는 함수 하나가 한 화면을 넘어갈 때. 재사용성 기준으로는 두 번 이상 사용될 코드는 함수로 만들고, 한 번만 쓰는 코드는 인라인 상태로 놔두는 것이다. **목적과 구현으로 분리하는 방식이 가장 합리적인 기준.** 코드를 보고 무슨 일을 하는지 파악하는데 한참이 걸린다면 그 부분을 함수로 추출한 뒤 ‘무슨 일’에 걸맞는 이름을 짓는다. 나중에 코드를 다시 읽을 때 함수의 목적이 눈에 확들어오고, 본문 코드(그 함수가 목적을 이루기 위해 구체적으로 수행하는 일)에 대해서는 더 이상 신경 쓸 일이 거의 없다.
- 켄트 벡이 보여준 스몰토크 시스템. 화면에서 텍스트나 그래픽을 강조하려면 해당 부분의 색상을 반전시켜야 했다. `highlight()` 메서드가 있었는데, 구현 코드를 보니 단순히 `reverse()`라는 메서드만 호출하고 있었다. 메서드 이름이 구현 코드보다 길었지만, 그건 문제가 되지 않았다. 코드의 목적(강조)과 구현(반전)사이의 차이가 그만큼 컸기 때문이다.
- 함수 호출이 많아져서 성능이 느려지는 것에 대해 걱정하지마라. 요즘은 그런일이 거의 없다. 함수가 짧으면 캐싱하기가 더 쉽기 때문에 컴파일러가 최적화하는데 더 유리할 때가 많다. 최적화할 때는 다음 두 규칙을 따르기 바란다. "The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet." — Michael A. Jackson
- 짧은 함수의 이름은 이를 잘 지어야만 발휘되므로 이를 짓기에 특별히 신경 써야 한다. 어느 정도 훈련이 필요하다.

함수 추출하기 (배경)



- "The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet." — Michael A. Jackson
- "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%." -
- Anonymous: I have the problems with many happy-coders that they only remember the part about not optimizing early and often forget that part when you should measure performance, find bottlenecks and get rid of them.



함수 추출하기 (절차)

1. 함수를 새로 만들고 목적을 잘 드러내는 이름을 붙인다.

대상 코드가 함수 호출 하나처럼 매우 간단하더라도 함수로 뽑아서 목적이 더 잘 드러나는 이름을 붙일 수 있다면 추출한다. 이름이 떠오르지 않는다면 함수로 추출하면 안 된다는 신호다. 처음부터 최선의 이름부터 짓고 시작 할 필요는 없고, 일단 함수로 추출해서 사용해보고 효과가 크지 않다면 다시 원래 상태로 인라인해도 된다. 그 과정에서 조금이라도 깨달은 게 있다면 시간 낭비는 아니다. 추출한 함수를 중첩 함수로 선언하면, 유효범위를 벗어난 변수를 처리하는 작업을 줄일 수 있다.

2. 추출할 코드를 원본 함수에서 복사하여 새 함수에 붙여넣는다.

3. 추출한 코드 중 원본 함수의 지역변수를 참조하거나 추출한 함수의 유효범위를 벗어나는 변수는 없는지 검사한다. 있다면 매개변수로 전달한다.

- 원본 함수의 중첩 함수로 추출할 때는 이런 문제가 생기지 않는다.
- 기존 함수에 지역 변수랑 매개변수로 받은 변수가 있다면 두 종류의 변수 모두 추출 함수의 인수로 전달한다. 값이 바뀌지 않는 변수는 대체로 이렇게 처리할 수 있다.
- 추출한 코드에서만 사용 되는 변수가 추출한 함수 밖에서 전달한다면, 함수 안으로 넣도록 한다.
- 추출한 코드 안에서 값이 바뀌는 변수 중에서 값으로 전달되는 것은 주의해서 처리한다. 이런 변수가 하나 뿐이라면 추출한 코드를 질의 함수로 취급해서 그 결과를 해당 변수에 대입한다.
- 때로는 추출한 코드에서 값을 수정하는 지역변수가 너무 많은 경우, 함수 추출을 멈추고 변수 쪼개기9.1이나 임시 변수를 질의 함수로 바꾸기7.4와 같은 다른 리팩터링을 적용해서 변수를 사용하는 코드를 단순하게 바꿔본다. 그 다음 계속 추출을 진행한다.

4. 변수를 다 처리했다면 컴파일한다.

5. 원본 함수에서 추출한 코드 부분을 새로 만든 함수를 호출하는 문장으로 바꾼다.

6. 테스트한다.

7. 다른 코드에 방금 추출한 것과 똑가거나 비슷한 코드가 없는지 살핀다. 있다면 방금 추출한 새 함수를 호출하도록 바꿀지 검토한다. 인라인코드를 함수 호출로 바꾸기8.5

9.1 변수 쪼개기

개요 AS IS

```
let temp = 2 * (height + width);
console.log(temp);
temp = height * width;
console.log(temp);

function discount(inputValue, quantity) {
  if (inputValue > 50) inputValue = inputValue - 2;
  if (quantity > 100) inputValue = inputValue - 1;
  return inputValue;
}
```

TO BE

```
const perimeter = 2 * (height + width);
console.log(perimeter);
const area = height * width;
console.log(temp);

function discount(originalInputValue, quantity) {
  let inputValue = originalInputValue;
  if (inputValue > 50) inputValue = inputValue - 2;
  if (quantity > 100) inputValue = inputValue - 1;
  return inputValue;
}

function discount(inputValue, quantity) {
  let result = inputValue;
  if (inputValue > 50) result = result - 2;
  if (quantity > 100) result = result - 1;
  return result;
}
```

1판에서 이름
임시변수 분리
매개변수로의 값 대입 제거

7.4 임시 변수를 질의 함수로 바꾸기

개요 AS IS

```
const basePrice = this._quantity * this._itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```

TO BE

```
const basePrice = () => this._quantity * this._itemPrice;
if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;
```

함수 추출하기 (예시)

1. 유효범위를 벗어나는 변수가 없을 때
간단히 추출 -> PrintBanner()
2. 지역 변수를 사용할 때
매개 변수로 넘기면 됨 -> printDetails()
3. 지역 변수의 값을 변경할 때
임시 변수를 새로 하나 만들어 그 변수에 대입 -> 변수쪼개기, 임시 변수를 질의함수로 바꾸기
반복문의 경우 변수가 사용되는 부분을 한곳에서 처리하도록 모아두기 -> 슬라이드

함수인라인

개요 AS IS

```
const rating =(aDriver) => {
    return moreThanFiveLateDeliveries(aDriver) ? 2 : 1;
}

const moreThanFiveLateDeliveries = (aDriver) => {
    return aDriver.number0fLateDelivereis > 5;
}

const reportLines = (aCustomer) => {
    const lines = []
    gatherCustomerData(lines, aCustomer);
    return lines;
}

const gatherCustomerData = (out, aCustomer) => {
    out.push(['name', aCustomer.name])
    out.push(['location', aCustomer.location])
}
```

TO BE

```
const rating =(aDriver) => {
    return (aDriver.number0fLateDelivereis > 5) ? 2 : 1;
}

const reportLines = (aCustomer) => {
    const lines = []
    lines.push(['name', aCustomer.name])
    lines.push(['location', aCustomer.location])
    return lines;
}
```

반대 리팩터링: 함수 추출하기

함수 인라인하기 (배경)

- 목적이 분명히 드러나는 이름의 짤막한 함수를 이용하기를 권한다. 그래야 코드가 명료해지고 이해하기 쉬워지기 때문이다. 하지만 함수 본문이 함수 이름 만큼 명확한 경우가 있다. 또는 함수 본문 코드를 이름만큼 깔끔하게 리팩터링할 때도 있다. 이럴 때는 그 함수를 제거한다.
- 잘못 추출된 함수들도 다시 인라인한다. 잘못 추출된 함수들을 원래 함수로 합친 다음 필요하면 원하는 형태로 다시 추출하는 것이다.
- 간접 호출을 너무 과하게 쓰는 코드도 흔한 인라인 대상이다. 가령 다른 함수로 단순히 위임하기만 함수들이 너무 많아서 위임 관계가 복잡하게 얹혀 있으면 인라인 해버린다.

함수 인라인하기 (절차)

- 다형 메서드인지 확인한다. 서브클래스에서 오버라이드하는 메서드는 인라인하면 안된다. => 서브클래스 함수를 슈퍼클래스로 올리지 마라.
- 인라인할 함수를 호출하는 곳을 모두 찾는다.
- 각 호출문을 함수 본문으로 교체한다.
- 하나씩 교체할 때마다 테스트한다. 인라인 작업을 한 번에 처리할 필요는 없다. 까다로운 부분이 있다면 일단 남겨두고 여유가 생길 때마다 틈틈히 처리하자.
- 함수 정의(원래 함수)를 삭제한다.

함수 인라인하기 (예시)

- rating 함수에서 moreThanFiveLateDeliveries 를 비교하는 예가 있는데, 이것만 보면 충분하게 사용가능 해보인다. 하지만 그리 복잡한게 아니라면 rating 하위로 바로 넣어도 되는 부분.
- Q. 만약 moreThanFiveLateDeliveries 가 50줄에 가까운 복잡한 연산 함수라면 추출하겠는가 인라인 하시겠는가요?
- 단순히 잘라붙이는 식으로 gatherCustomerData를 옮길 수 없다면, 한 줄 한 줄을 reportLines에 넣고 테스트를 반복하면서 수행한다. 핵심은 단계를 잘게 쪼개서 처리하는데에 있다. 어떨때는 함수를 작게 만들어 뒀다면 인라인을 단번에 처리할 수 있을 때가 많지만, 복잡한 경우 한 번에 한 문장씩 처리한다. 한 문장을 처리하더라도 얼마든지 복잡해질 수 있다.

```
const rating =(aDriver) => {
  return moreThanFiveLateDeliveries(aDriver) ? 2 : 1;
}

const moreThanFiveLateDeliveries = (aDriver) => {
  return aDriver.number0fLateDelivereis > 5;
}
```

```
const reportLines = (aCustomer) => {
  const lines = []
  gatherCustomerData(lines, aCustomer);
  return lines;
}

const gatherCustomerData = (out, aCustomer) => {
  out.push(['name', aCustomer.name])
  out.push(['location', aCustomer.location])
}
```