테스트 커버리지에 현혹되지 말자 Software Quality

얼마 전에 열렸던 회사 컨퍼런스에서 테스트 커버리지를 주제로 한 발표가 있었다. 발표 내용을 간략하게 요약하면 어떤 서비스에 단위 테스트를 작성하도록 정책적인 장치를 마련한 결과 70%의 테스트 커버리지를 얻게 되었다는 것이다. 코드에 실행 가능한 단위 테스트가 존재하고 팀이 지속적으로 70%의 테스트 커버리지를 유지할 수 있다는 것은 팀과 코드의 상태 모두 양호하고 프로젝트의 가시성이 확보되었다는 것을 나타내주는 긍정적인 척도다. 그러나 문제는 발표에서 언급하고 있는 테스트 커버리지의 종류가 라인 커버리지(Line Coverage)라는데 있다. 그리고 더 큰 문제는 대부분의 사람들이 라인 커버리지라는 용어는 무시한 채 70%라는 수치 자체를 중요시 한다는데 있다.

결론부터 말하자면 테스트 커버리지에 대한 정책을 수립할 때 라인 커버리지(Line Coverage)나 브랜치 커버리지(Branch Coverage)를 목표로 하는 것은 옳지 않다. 특히 앞의 예에서처럼 테스트 커버리지에 대한 전반적인 이해 없이 맹목적으로 테스트 커버리지 수치만을 강조할 경우 오히려 코드 전체의 품질이 저하될 위험이 있다. 뒤에서 살펴보겠지만 70%의 라인 커버리지를 달성했다고 해서 70%의 코드 품질을 달성했다고 장담할 수는 없다. 따라서 정책적으로 무의미한 목표 커버리지를 강요할 경우 개발자가 취할 수 있는 마지막 방법은 코드의 품질을 보장하지는 않더라도 테스트 커버리지를 높일 수 있는 무가치한 테스트 케이스를 작성하는 것이다.

라인 커버리지(Line Coverage)-또는 스테이트먼트 커버리지(Statement Coverage)라고도 한다-는 전체 코드 중 테스트에 의해 실행된 코드 라인 수의 비율을 의미한다. 전체 100라인의 코드 중에서 테스트 케이스가 80 라인을 실행한 경우의 라인 커버리지는 80%다.

어떤 개념을 이해하는 가장 좋은 방법은 실행되는 예제를 보는 것이므로 판매 도메인과 관련된 간단한 예제를 통해 라인 커버리지의 문제점을 살펴 보도록 하자. 어떤 상점에서 고객에게 물건을 판매할 때 고객의 등급과 상품의 할인 여부에 따라 할인율을 다르게 적용한다고 가정하자. 고객 등급은 VIP, GOLD, SILVER, REGULAR로 나뉘며 VIP 회원일 경우 0.1%, GOLD 회원일 경우 0.05%, SILVER회원일 경우 0.02%, REGULAR 회원일 경우에는 할인되지 않는다. 고객 등급과 별도로 상품이 할인 가능한 경우 상품에 할당된 할인율에 따라 가격

을 할인해 준다.

고객 등급과 등급별 할인율은 Credit 열거형에 정의한다. discountAmount() 메소드는 고객 등급에 따라 할인된 가격을 계산하여 반환한다.

Credit.java

```
public enum Credit {

VIP(0.1), GOLD(0.05), SILVER(0.02), REGULAR(0);


private double discountRate;


private Credit(double discountRate) {

this.discountRate = discountRate;

}


public Money discountAmount(Money price) {

return price.minus(price.times(discountRate));

}

}
```

Product은 상품의 명칭, 가격, 할인율을 속성으로 가지고, 상품 자체의 할인율에 따라 할인된 가격을 계산한다.

Product.java

```
public class Product {

private String name;

private Money price;

private double discountRate;


Product(String name, Money price) {

this.name = name;

this.price = price;

this.discountRate = 0;

}


Product(String name, Money price, double discountRate) {

this(name, price);

this.discountRate = discountRate;

}


Money getPrice() {
```

```
return price;
}

public boolean isDiscountable() {
return discountRate != 0;
}

public Money discount(Money amount) {
return amount.minus(amount.times(discountRate));
}
}
```

Customer는 이름과 고객 등급을 속성으로 가진다.

Customer.java

```
public class Customer {
private Credit credit;
private String name;
```

```
public Customer(String name, Credit credit) {

this.name = name;

this.credit = credit;

}


public boolean isSpecialCredit() {

return credit != Credit.REGULAR;

}


public Credit getCredit() {

return credit;

}

}
```

우리가 테스트하려는 대상은 Seller 클래스의 calculatePrice() 메소드로 Customer, Product, 상품의 할인 여부를 선택하는 applyProductDiscount 속성을 받아 금액을 계산한다. 고객 등급에 따른 할인 정책은 항상 적용되지만 상품 자체의 할인율은applyProductDiscount 파라미터가 **true** 인 경우에만 적용된다(이 예제는 INFORMATION EXPERT도 준수하지 않고 여기 저기 책임도 잘못 할당되어 있다. 설명을 위해 인위적으로 만든 예제라고 생각하고 봐주기 바란다).

Seller.java

```
Money calculatePrice(Customer customer, Product product,

boolean applyProductDiscount) {

Money amount = product.getPrice();


if (customer.isSpecialCredit()) {

amount = customer.getCredit().discountAmount(amount);

}


if (applyProductDiscount) {

amount = product.discount(amount);

}


return amount;

}
```

라인 커버리지는 단순히 테스트 케이스에 의해 실행된 calculatePrice() 메소드의 라인 수를 측정하기 때문에 두 조건식을 **true**로 만드는 파라미터를 전달하면 간단하게 100%의 라인 커버리지를 얻게 된다.

SellerTest.java

```
@Test

public void calculateVIPCustomerAmount() {

Customer customer = new Customer("조영호", Credit.VIP);

Product product = new Product("X-Box 360", Money.wons(300000), 0.05);



assertEquals(Money.wons(256500),

new Seller().calculatePrice(customer, product, true));

}
```

테스트 케이스 하나로 100%의 라인 커버리지를 얻었지만 두 **if** 문 내의 조건식이 **false** 인 경우는 테스트하지 않았다. 결론적으로 100%의 라인 커버리지를 얻는 것은 매우 간단하지만 코드의 품질을 보장하기에는 신뢰성이 떨어진다는 사실을 알 수 있다.

브랜치 커버리지(Branch Coverage)는 라인 커버리지의 단점을 보완하고 개별 조건식이 **true**, **false** 인 경우 모두 테스트되었는지를 판단하는 테스트 커버리지 유형이다. 앞의 예에서 고객이 특별 등급인 경우와 아닌 경우(두 조건의 조합이 TRUE-TRUE인 경우),상품의 할인 여부를 적용하는 경우와 적용하지 않는 경우(두 조건의 조합이 FALSE-FLASE인 경우)의 테스트 케이스가 모두 존재할 경우 100%의 브랜치 커버리지를 얻을 수 있게 된다.

앞에서 작성한 테스트 케이스는 고객이 특별 등급인 경우와 상품 할인 여부를 적용하는 경우를 테스트하므로, 고객이 특별 등급이 아닌 경우와 상품 할인 여부를 적용하지 않는 경우에 대한 테스트 케이스를 추가하면 100%의 브랜치 커버리지를 얻을 수 있다.

SellerTest.java

```
@Test
public void calculateRegularCustomerAmount() {
Customer customer = new Customer("조영호", Credit.REGULAR);
Product product = new Product("X-Box 360", Money.wons(300000));

assertEquals(Money.wons(300000),
new Seller().calculatePrice(customer, product, false));
}
```

테스트 커버리지를 측정할 수 있는 다양한 오픈 소스, 상용 제품이 존재하며, Ant, Maven과 같은 빌드 스크립트와 CI 툴에 테스트 커버리지 측정 도구를 플러그인할 수 있다. 여기에서는 오픈 소스 Cobertura를 사용해서 라인 커버리지와 브랜치 커버리지를 측정한 결과를 표시한 것이다. 두 개의 테스트 케이스를 실행한 결과 Seller의 라인 커버리지와 브랜치 커버리지 모두 100%를 달성한 것에 주목하자.

Coverage Report - org.eternity.selling

Package	# Classes	Line Coverage	Branch Coverage	Complexity
org.eternity.selling	4	97% 29/30	75% 6/8	1.222

Classes in this Package	Line Coverage	Branch Coverage	Complexity
Credit	100% 6/6	N/A N/A	0
Customer	100% 6/6	100% 2/2	1
Product	91% 10/11	0% 0/2	1
Seller	100% 7/7	100% 4/4	3

브랜치 커버리지의 경우 개별 조건의 TRUE, FALSE 여부가 테스트 되었는지는 확인 가능하

지만 조건의 조합이 테스트되었는지는 확인할 수 없다. 따라서 고객이 일반 회원이고 제품의 할인율을 적용하는 경우나, 고객이 특별 회원이고 제품의 할인율을 적용하지 않는 경우는 테스트하지 않는다는 단점이 있다. 따라서 라인 커버리지와 브랜치 커버리지 모두 100%를 달성했다고 하더라도 코드 자체의 모든 경로를 테스트했다고 볼 수는 없다.

이런 단점을 보완하기 위해서는 패스 커버리지(Path Coverage)를 목표로 삼아야 한다. 패스 커버리지는 개별 라인이나 개별 조건식이 아닌 메소드가 실행될 수 있는 경로의 조합을 테스트하는 방식을 사용한다. 일반적으로 패스 커버리지를 만족시키기 위해서는 최소 순환 복잡도(Cyclomatic Complexity)만큼의 테스트 케이스가 필요하다. 위 Cobertura 레포트에서 Complexity 컬럼이 바로 순환 복잡도를 의미한다. 따라서 100%의 패스 커버리지를 달성하기 위해서는 최소 3개의 테스트 케이스가 필요하다.

패스 커버리지를 만족시키는 테스트 케이스의 조합을 만들기 위해서는 간단하게 개별 조건식의 결과를 순서대로 토글시키도록 파라미터를 전달하면 된다. 판매 예에서는 고객의 특별 등급 여부와 상품 할인 여부에 대한 두 개의 조건식이 존재하므로, 고객이 특별 등급이고 상품 할인을 적용하는 경우(TRUE-TRUE)와 고객이 특별 등급이 아니고 상품 할인을 적용하는 경우(FALSE-TRUE), 마지막으로 고객이 특별 등급이고 상품 할인을 적용하지 않는 경우(TRUE-FALSE)의 3가지 테스트 케이스를 작성하면 된다.

SellerTest.java

```
@Test
public void calculateVIPCustomerAmount() {
Customer customer = new Customer("조영호", Credit.VIP);
Product product = new Product("X-Box 360", Money.wons(300000), 0.05);

assertEquals(Money.wons(256500),
```

```
    new Seller().calculatePrice(customer, product, true));
}

@Test
public void calculateRegularCustomerAndDiscountAmount() {
Customer customer = new Customer("조영호", Credit.REGULAR);
Product product = new Product("X-Box 360", Money.wons(300000), 0.05);

assertEquals(Money.wons(285000),
    new Seller().calculatePrice(customer, product, true));
}

@Test
public void calculateVIPCustomerAndNonDiscountAmount() {
Customer customer = new Customer("조영호", Credit.VIP);
Product product = new Product("X-Box 360", Money.wons(300000), 0.05);

assertEquals(Money.wons(270000),
    new Seller().calculatePrice(customer, product, false));
}
```

지금까지 살펴본 바와 같이 팀에서 테스트 커버리지를 측정하고자 한다면 라인 커버리지와 브랜치 커버리지가 아니라 패스 커버리지를 목표로 해야 한다. 물론 라인 커버리지에서 패스 커버리지로 단계적으로 목표를 조정해 가는 것은 좋은 방법이지만 맹목적으로 라인 커버리지를 강제하는 것은 올바른 접근 방법이 아니다. 모든 클래스가70%의 라인 커버리지를 갖는 것보다 소수의 핵심 클래스가 70%의 패스 커버리지를 갖는 것이 더 좋은 품질을 보장할 수 있다는 점에 주목하자.

파레토 원리를 잊지 말자. 우리는 전체 오류의 80%를 차지하는 20% 부분에 집중해야 한다. 테스트 커버리지 측정은 적은 노력으로도 매우 큰 이익을 얻을 수 있는 효과적인 프랙티스다. 테스트 커버리지를 바보들의 황금으로 만드는 것은 테스트 그 자체가 아니라 테스트 커버리지를 사용하는 우리들 자신이라는 것을 잊지 말아야 한다. 배를 난파시키려는 세이렌의 노래에 현혹되지 말자. 눈을 감고 귀를 막을 필요까지는 없지만 무엇이 옳은 길인가에 대한 냉철한 판단력을 유지해야 한다.