

Chap11. 단위 테스트의 실제

이번 장의 목표

- 코드의 모든 동작을 효과적이고 신뢰성 있게 테스트하기
- 이해하기 쉽고 실패가 잘 설명되는 테스트 코드의 작성
- 의존성 주입을 사용하여 테스트가 용이한 코드의 작성

들어가며

이전 장에서 설명한 좋은 단위 테스트의 주요 특징을 다시 한번 상기시키며, 이러한 특징을 충족하기 위해 적용 가능한 실제적인 방법들을 제안합니다. 좋은 단위 테스트의 주요 특징은 다음과 같습니다:

1. 문제를 정확하게 감지:

- 코드에 문제가 있으면 테스트는 실패해야 하며, 실제로 문제가 있을 때만 테스트가 실패해야 합니다.

2. 구현 세부 사항에 구애받지 않음:

- 코드의 구현 세부 사항이 변경되더라도 테스트 코드에는 변화가 없어야 하는 것이 이상적입니다.

3. 실패 시 명확한 설명 제공:

- 코드에 문제가 있을 경우, 테스트 실패는 문제에 대한 명확한 설명을 제공해야 합니다.

4. 이해하기 쉬운 테스트 코드:

- 테스트가 무엇을 수행하며, 어떻게 작동하는지를 다른 개발자가 이해할 수 있어야 합니다.

5. 쉽고 빠르게 실행 가능:

- 단위 테스트는 자주 실행되어야 하므로, 실행이 쉽고 빠르게 이루어져야 합니다. 느리거나 실행이 어렵다면 개발자의 시간이 낭비될 수 있습니다.

이러한 특징을 갖추도록 테스트를 작성하는 것은 쉽지 않으며, 비효율적이거나 유지보수하기 어려운 테스트 코드를 작성하기 쉬운 이유이기도 합니다. 다행히도 이러한 특징을 구현하기 위한 실질적인 방법들이 많이 있으며, 이 장에서는 그중 몇 가지 주요 방법들을 소개합니다.

11.1 기능뿐만 아니라 동작을 시험하라

코드를 테스트하는 것은 작업 목록을 만들어 체크하는 것과 유사합니다. 테스트할 코드가 수행하는 여러 작업이 있다면, 각 작업에 대해 별도의 테스트 케이스를 작성해야 합니다. 그러나 단순히 함수 이름만 테스트 목록

에 추가하는 실수를 종종 하게 됩니다. 예를 들어, 클래스에 두 개의 함수가 있으면 각 함수에 대해 하나씩 테스트 케이스를 작성하는 식입니다.

코드의 중요한 행동을 모두 테스트해야 한다는 원칙에 따르면, 각 함수에 대해 단순히 하나의 테스트 케이스만 작성해서는 안 됩니다. 왜냐하면 한 함수가 여러 동작을 수행할 수도 있고, 하나의 동작이 여러 함수에 걸쳐 있을 수 있기 때문입니다. 함수별로 테스트 케이스를 하나만 작성하면 중요한 동작을 놓칠 가능성이 큽니다.

Note

따라서, 테스트 목록에는 함수 이름만 나열하는 대신, 함수가 수행하는 모든 동작을 기준으로 테스트 목록을 작성하는 것이 좋습니다.

11.1.1 함수당 하나의 테스트 케이스만 있으면 적절하지 않을 때가 많다

예제 11.1 주택담보대출 평가 코드

```
class MortgageAssessor {  
    private const Double MORTGAGE_MULTIPLIER = 10.0;  
  
    MortgageDecision assess(Customer customer) {  
        if (!isEligibleForMortgage(customer)) {  
            return MortgageDecision.rejected();  
        }  
        return MortgageDecision.approve(getMaxLoanAmount(customer));  
    }  
  
    private static Boolean isEligibleForMortgage(Customer customer) {  
        return customer.hasGoodCreditRating() &&  
            !customer.hasExistingMortgage() &&  
            !customer.isBanned();  
    }  
  
    private static MonetaryAmount getMaxLoanAmount(Customer customer) {  
        return customer.getIncome()  
            .minus(customer.getOutgoings())  
            .multiplyBy(MORTGAGE_MULTIPLIER);  
    }  
}
```

고객이 자격이 없다면
대출 신청은 거절된다.

고객이 자격이 있는지 여부를 결정하는
프라이빗 헬퍼 함수

대출의 최대 액수를 결정하는
프라이빗 헬퍼 함수

은행의 주택 담보 대출 신청 자동 평가 시스템을 운영하는 상황을 가정해 보겠습니다. 이 시스템에서 고객의 대출 자격과 대출 가능 금액을 결정하는 클래스가 있습니다. 이 클래스의 `assess()` 함수는 다음과 같은 역할을 수행합니다.

1. 대출 자격 평가:

- `assess()` 함수는 프라이빗 헬퍼 함수를 호출하여 고객이 대출 자격이 있는지 판단합니다. 고객이 자격을 얻기 위해서는 다음 조건을 충족해야 합니다.
 - 신용 등급이 좋아야 합니다.
 - 기존에 주택 담보 대출이 없어야 합니다.
 - 은행에 의해 금지된 고객이 아니어야 합니다.

2. 대출 금액 계산:

- 고객이 자격이 있는 경우, 또 다른 프라이빗 헬퍼 함수가 호출되어 최대 대출 금액이 계산됩니다. 이 금액은 고객의 연간 소득에서 연간 지출을 뺀 후 10을 곱한 값입니다.
- 예제 테스트 코드에서는 `assess()` 함수에 대해 하나의 테스트 케이스만 존재하며, 이 테스트는 다음 두 가지 조건을 확인합니다.
 - 신용 등급이 좋고 기존 대출이 없으며 금지된 고객이 아닌 경우 대출이 승인되는지
 - 최대 대출 금액이 고객의 수입에서 지출을 뺀 값에 10을 곱한 금액인지

예제 11.2 주택 담보 대출 평가 테스트

```
testAssess() {  
    Customer customer = new Customer(  
        income: new MonetaryAmount(50000, Currency.USD),  
        outgoings: new MonetaryAmount(20000, Currency.USD),  
        hasGoodCreditRating: true,  
        hasExistingMortgage: false,  
        isBanned: false);  
    MortgageAssessor mortgageAssessor = new MortgageAssessor();  
  
    MortgageDecision decision = mortgageAssessor.assess(customer);  
  
    assertThat(decision.isApproved()).isTrue();  
    assertThat(decision.getMaxLoanAmount()).isEqualTo(  
        new MonetaryAmount(300000, Currency.USD));  
}
```

그러나 이 테스트 케이스는 대출이 거절되는 경우 등 다양한 상황을 다루지 않아 충분하지 않습니다. 심지어 금지된 고객에게도 대출이 승인되도록 `assess()` 함수를 수정하더라도 테스트는 여전히 통과할 수 있습니다.

```

private static Boolean isEligibleForMortgage(Customer customer) {
    return customer.hasGoodCreditRating() &&
        !customer.hasExistingMortgage(); // `!customer.isBanned()` 조건이
제거됨
}

void testAssess() {
    Customer customer = new Customer(
        income: new MonetaryAmount(50000, Currency.USD),
        outgoings: new MonetaryAmount(20000, Currency.USD),
        hasGoodCreditRating: true,
        hasExistingMortgage: false,
        isBanned: false // 금지된 고객이 아님
    );

    MortgageAssessor mortgageAssessor = new MortgageAssessor();
    MortgageDecision decision = mortgageAssessor.assess(customer);

    assertThat(decision.isApproved()).isTrue();
    assertThat(decision.getMaxLoanAmount()).isEqualTo(
        new MonetaryAmount(300000, Currency.USD)
    );
}

```

하지만 테스트는 대출 승인 여부에만 초점을 두고, 거절 조건 처리에 집중하지 않았기에 테스트는 통과합니다. 원래 `assess()` 함수는 금지된 고객에게 대출이 승인되지 않도록 설계되어야 하지만, 금지된 고객에게도 대출이 승인되도록 잘못 설정된 경우에도 테스트가 여전히 통과하는 상황이 발생하는 상황입니다.

이러한 문제는 테스트를 작성하는 개발자가 **기능(대출 승인 여부)**에만 초점을 두고, **행동(거절 조건 처리 등)**에 집중하지 않았기 때문입니다. `assess()` 함수가 `MortgageAssessor` 클래스의 유일한 퍼블릭 API이기 때문에 하나의 테스트 케이스만 작성했지만, 이 테스트만으로는 `assess()` 함수의 올바른 동작을 확인하기에 충분하지 않습니다.

Note

앞 10장에서 말했듯이 퍼블릭 API를 테스트하는게 원칙이지만, public API 만으로는 올바르게 동작을 하는지 확인할 수 없는 경우가 해당 사례로 보입니다!

11.1.2 해결책: 각 동작을 테스트하는데 집중하라

함수와 동작이 항상 일대일로 연결되지 않는 경우가 많기 때문에, 함수 자체만을 테스트하면 실제 중요한 동작을 검증하지 못하는 테스트 케이스를 작성하기 쉽습니다. `MortgageAssessor` 클래스의 중요한 동작들은 다음과 같습니다.

1. 대출 거절 조건:

- 다음 조건 중 하나라도 해당하는 고객에 대해서는 대출 신청이 기각되어야 합니다.
 - 신용 등급이 좋지 않은 경우
 - 기존에 대출이 있는 경우
 - 금지된 고객인 경우

2. 대출 승인 및 최대 대출 금액:

- 주택 담보 대출 신청이 승인된 경우, 최대 대출 금액은 고객의 연간 소득에서 연간 지출을 뺀 금액에 10을 곱한 값이 되어야 합니다.

이러한 동작을 모두 테스트 하려면 다양한 테스트 케이스가 필요하며, 테스트 코드의 신뢰성을 높이기 위해 여러 값과 경계 조건을 테스트하는 것도 중요합니다.

- 소득과 지출에 대해 다양한 값을 사용하여 계산이 정확한지 확인하는 테스트
- 소득이나 지출이 없는 경우, 혹은 소득이나 지출이 매우 많은 극단적인 경우에 대한 테스트

`MortgageAssessor` 클래스의 모든 동작을 완전히 검증하려면 다양한 상황에 대한 10개 이상의 테스트 케이스가 필요할 수 있습니다.

- 이는 정상적인 일이며, 실제 코드가 100줄 정도라면 테스트 코드가 300줄에 달하는 경우도 흔하다고 합니다. 오히려 테스트 코드의 양이 실제 코드보다 적다면, 필요한 모든 동작이 충분히 검증되지 않았다는 신호일 수 있을 수 있다고 합니다.

테스트할 동작을 고민하는 과정에서 코드의 숨겨진 문제를 발견하기도 합니다. 예를 들어, 고객의 지출이 수입을 초과하는 상황을 생각해 보면, 현재 `MortgageAssessor.assess()` 함수가 이런 경우에도 대출을 승인하고 최대 대출 금액을 음수로 계산할 가능성이 있습니다. 이런 비정상적인 결과를 방지하려면, 해당 상황에 맞는 추가적인 논리 검토와 개선이 필요합니다.

모든 동작이 테스트되었는지 거듭 확인하라

코드가 제대로 테스트되고 있는지 확인하는 좋은 방법은 코드에 버그나 오류가 있어도 테스트가 통과할 수 있는지를 생각해 보는 것입니다. 코드 검토 과정에서 다음과 같은 질문을 던져볼 수 있습니다. 만약 이 중 하나라도 “예”라고 답할 수 있다면, 모든 동작이 테스트되지 않았을 가능성이 큽니다.

- 삭제해도 컴파일되거나 테스트가 통과하는 코드가 있는가?
- `if` 문 등의 조건문에서 참/거짓을 반대로 해도 테스트가 통과하는가? (예: `if (something)`을 `if (!something)`으로 변경)
- 논리 연산자나 산술 연산자를 다른 것으로 대체해도 테스트가 통과하는가? (예: `&&`를 `||`로, `+`를 `-`로 변경)
- 상수값이나 하드코딩된 값을 변경해도 테스트가 통과하는가?

각 코드 줄, 조건문, 논리 표현식, 값 등은 그 존재 이유가 있어야 합니다. 불필요한 코드라면 제거해야 하며, 그렇지 않다면 중요한 동작이 해당 코드에 의존하고 있다는 뜻입니다. 따라서 코드의 중요한 동작이 있다면 이를 검증하는 테스트 케이스가 필요하며, 기능이 변경되면 적어도 하나의 테스트 케이스는 실패해야 합니다. 그렇지 않다면 모든 동작이 테스트된 것이 아닙니다.

단, 예외적으로 방어적인 프로그래밍을 위해 작성된 오류 검사는 테스트하기 어려울 수 있습니다. 예를 들어, 특정 조건이 유효한지 확인하기 위해 코드에 `check`나 `assertion`을 사용할 수 있습니다. 이러한 방어 논리는 코드를 의도적으로 잘못 수정해 보지 않는 한 테스트하기 어려운 경우가 있습니다.

이를 자동화하는 방법 중 하나가 **돌연변이 테스트(mutation testing)**입니다. 돌연변이 테스트 도구는 코드에 약간의 변형을 가한 후, 변형된 코드로도 테스트가 통과하는지 확인합니다. 만약 변형된 코드로도 테스트가 통과한다면, 모든 동작이 제대로 테스트되지 않고 있다는 신호가 됩니다.

오류 시나리오를 잊지 말라

테스트에서 종종 간과되는 중요한 부분 중 하나는 오류 시나리오가 발생했을 때 코드가 어떻게 작동하는지입니다. 오류가 자주 발생하지 않을 것이라 예상하기 때문에 이러한 시나리오는 경계 조건처럼 놓치기 쉽습니다. 하지만 코드가 다양한 오류 시나리오를 어떻게 처리하고 알리는지는 작성자와 호출자 모두에게 중요한 동작이므로 반드시 테스트되어야 합니다.

예제 11.3 오류를 처리하는 코드

```
class BankAccount {  
    ...  
    void debit(MonetaryAmount amount) {  
        if (amount.isNegative()) {  
            throw new ArgumentException("액수는 0보다 적을 수 없음");  
        }  
        ...  
    }  
}
```

음수값이면
ArgumentException을
발생시킨다.

예제 11.4 오류 처리 테스트

```
void testDebit_negativeAmount_throwsArgumentException {  
    MonetaryAmount negativeAmount =  
        new MonetaryAmount(-0.01, Currency.USD);  
    BankAccount bankAccount = new BankAccount();  
  
    ArgumentException exception = assertThrows(  
        ArgumentException,  
        () -> bankAccount.debit(negativeAmount));  
    assertThat(exception.getMessage())  
        .isEqualTo("액수는 0보다 적을 수 없음");  
}
```

debit() 함수가 음수값으로 호출되면
ArgumentException이 발생하는지 확인한다.

발생한 예외 객체가 예상하는 오류 메시지를
가지고 있는지 확인한다.

예를 들어, BankAccount.debit() 함수가 음수 값을 인수로 받으면 ArgumentException을 발생시킵니다. 음수 값은 오류 시나리오에 해당하며, 이 경우 ArgumentException이 발생해야 한다는 것은 중요한 동작이므로 반드시 테스트 되어야 합니다.

코드 예시 11.4에서는 이러한 오류 시나리오를 테스트하는 방법을 보여줍니다. 이 테스트 케이스는 -0.01의 금액으로 debit()을 호출할 때 예외가 발생하는지 확인하며, 발생한 예외에 예상된 오류 메시지가 포함되어 있는지도 검증합니다.

하나의 함수는 다양한 동작을 나타낼 수 있습니다. 호출되는 값이나 시스템 상태에 따라 함수의 동작이 달라질 수 있기 때문에, 함수당 하나의 테스트 케이스만 작성하는 것은 충분하지 않습니다. 함수 자체에만 집중하기보다는, 중요한 모든 동작을 파악하고 각 동작에 대해 테스트 케이스가 있는지 확인하는 것이 더 효과적입니다.

11.2 테스트만을 위해 퍼블릭으로 만들지 말라

```

class MyClass {
    String publicFunction() { ... }           ← 클래스 외부에서도 보인다.

    private String privateFunction1 { ... }
    private String privateFunction2 { ... }     ← 클래스의 내부에서만 보인다.

}

```

클래스(또는 코드 단위)는 일반적으로 외부에서 사용할 수 있는 **퍼블릭(public)** 함수를 가지고 있습니다. 이 퍼블릭 함수들이 모여 코드의 공개 API를 구성합니다. 반면에, 클래스 내부에서만 사용되는 **프라이빗(private)** 함수도 있으며, 이는 외부 코드에서 직접 접근할 수 없습니다.

프라이빗 함수는 구현 세부 사항에 속하므로 클래스 외부에서 인지하거나 직접 사용할 필요가 없습니다. 가끔 프라이빗 함수에 접근할 수 있도록 만들어 테스트 코드에서 직접 테스트하고 싶어질 때가 있습니다. 하지만 이는 좋은 방법이 아닐 수 있습니다. 프라이빗 함수는 구현에 밀접하게 연관되어 있기 때문에, 이를 테스트하는 것은 코드의 구체적인 동작이 아닌 구현 세부 사항에 집중하는 테스트가 될 위험이 있기 때문입니다.

11.2.1 Private 함수를 테스트하는건 바람직 하지 않을 때가 많다

예제 11.5 프라이빗 헬퍼 함수가 있는 클래스

```

class MortgageAssessor {
    ...

    MortgageDecision assess(Customer customer) { ... }           ← 퍼블릭 API

    private static Boolean isEligibleForMortgage(
        Customer customer) { ... }

    private static MonetaryAmount getMaxLoanAmount(
        Customer customer) { ... }

}

```

앞서 MortgageAssessor 클래스의 모든 동작을 테스트하는 것이 중요하다고 살펴봤습니다. 이 클래스의 퍼블릭 함수는 `assess()` 함수이고, 내부에는 `isEligibleForMortgage()` 와 `getMaxLoanAmount()` 라는 두 가지 `private` 헬퍼 함수가 있습니다. 이 `private` 함수들은 클래스 외부에서 접근할 수 없기 때문에 구현 세부 사항에 해당합니다.

예를 들어, 고객의 신용 등급이 나쁠 경우 대출이 거부되는 동작을 테스트하려고 할 때, `isEligibleForMortgage()` 가 거짓을 반환하는지 확인하는 것이 떠오를 수 있습니다. 하지만 이 `private` 함수를 테스트 코드에서 접근 가능하게 만들고 싶어지는 것은 바람직하지 않습니다.

예제 11.6 프라이빗 함수를 퍼블릭으로 만듬

```
class MortgageAssessor {  
    private const Double MORTGAGE_MULTIPLIER = 10.0;  
  
    MortgageDecision assess(Customer customer) { ← 퍼블릭 API  
  
        if (!isEligibleForMortgage(customer)) { ← 어떤 헬퍼 함수가 호출될지는  
            return MortgageDecision.rejected(); 구현 세부 사항이다.  
        }  
        return MortgageDecision.approve(getMaxLoanAmount(customer));  
    }  
  
    /** 테스트를 위해서만 공개 */  
    static Boolean isEligibleForMortgage(Customer customer) { ← 퍼블릭으로 만들어져  
        return customer.hasGoodCreditRating() && 직접 테스트할 수 있다.  
            !customer.hasExistingMortgage() &&  
            !customer.isBanned();  
    }  
  
}
```

예제 11.7 프라이빗 함수 테스트

```
testIsEligibleForMortgage_badCreditRating_ineligible() {  
    Customer customer = new Customer(  
        income: new MonetaryAmount(50000, Currency.USD),  
        outgoings: new MonetaryAmount(25000, Currency.USD),  
        hasGoodCreditRating: false,  
        hasExistingMortgage: false,  
        isBanned: false);  
  
    assertThat(MortgageAssessor.isEligibleForMortgage(customer)) ← '프라이빗' 함수인  
        .isFalse(); isEligibleForMortgage()를  
    } 직접 테스트한다.
```

프라이빗 함수를 테스트하려고 퍼블릭으로 변경할 때의 문제점

1. 중요한 동작을 테스트하지 않게 된다: 고객의 신용 등급이 낮을 때 대출이 거절되는 것이 중요한 결과입니다. 그러나 `isEligibleForMortgage()` 를 테스트하는 경우, 실제로 중요한 것은 대출 거부 여부인데, 이 테스트는 `isEligibleForMortgage()` 함수가 거짓을 반환하는지만 확인합니다. 만약 `assess()` 함수가 잘못 수정되어 이 함수를 호출하지 않게 된다면, 대출이 거부되지 않더라도 테스트는 통과하게 됩니다.
2. 테스트가 구현 세부 사항에 의존하게 된다: `isEligibleForMortgage()` 함수는 구현 세부 사항이며, 리팩터링 과정에서 함수 이름이 바뀌거나 다른 헬퍼 클래스로 옮겨질 수 있습니다. 그러나 이 프라이빗 함수를 직접 테스트하고 있으면, 리팩터링 시 불필요한 테스트 실패가 발생할 수 있습니다.
3. 퍼블릭 API가 불필요하게 확장된다: ‘테스트를 위해서만 공개’라는 주석을 추가하더라도, 다른 개발자가 이 함수를 호출하고 의존할 가능성이 생깁니다. 이렇게 되면 다른 코드가 이 함수에 의존하게 되어, 리팩터링이나 수정이 어려워질 수 있습니다.

좋은 단위 테스트는 중요한 동작을 테스트해야 하며, 이를 통해 테스트는 코드의 문제를 정확히 감지하고 구현 세부 사항에 독립적일 수 있습니다. 이 두 가지는 이상적인 단위 테스트의 중요한 특징입니다. 프라이빗 함수를 테스트하면 이러한 원칙을 지키기 어렵습니다.

프라이빗 함수를 테스트하지 않고도 퍼블릭 API를 통해 테스트하거나 코드를 적절히 추상화하여 문제를 해결 할 수 있습니다.

11.2.2 해결책: Public API를 통해 테스트 하라

이전 장에서 논의한 “퍼블릭 API만을 이용한 테스트” 원칙은 개발자가 구현 세부 사항이 아닌, 실제로 중요한 동작을 테스트하도록 안내합니다. 프라이빗 함수가 퍼블릭으로 공개되어 있다면, 이는 이 원칙이 지켜지지 않고 있다는 경고 신호로 볼 수 있습니다.

예제 11.8 퍼블릭 API를 통한 테스트

```
testAssess_badCreditRating_mortgageRejected() {
    Customer customer = new Customer(
        income: new MonetaryAmount(50000, Currency.USD),
        outgoings: new MonetaryAmount(25000, Currency.USD),
        hasGoodCreditRating: false,
        hasExistingMortgage: false,
        isBanned: false);
    MortgageAssessor mortgageAssessor = new MortgageAssessor();

    MortgageDecision decision = mortgageAssessor.assess(customer);

    assertThat(decision.isApproved()).isFalse();
}
```

퍼블릭 API를 통해 행동을 테스트한다.

`MortgageAssessor` 클래스의 경우 중요한 동작은 신용 등급이 낮은 고객의 주택 담보 대출 신청을 거절하는 것입니다. 이 동작은 `MortgageAssessor` 클래스의 퍼블릭 API인 `assess()` 함수를 통해 테스트할 수

있습니다. 예제 11.8은 `assess()` 함수를 통해 이 동작을 테스트하는 코드를 보여줍니다. 이처럼 퍼블릭 API를 통해 실제로 중요한 동작을 테스트하면, 프라이빗 함수를 퍼블릭으로 만들 필요가 없어집니다.

실용적으로 하라

테스트를 위해 프라이빗 함수를 퍼블릭으로 만들어 외부로 보이게 하는 것은 대부분의 경우 구현 세부 사항을 테스트한다는 것을 보여주는 경고 신호이며, 일반적으로 더 나은 대안이 있다. 그러나 ‘퍼블릭 API만을 이용한 테스트’라는 원칙을 다른 것(예: 의존성)에 적용할 때는 10장(10.3절)의 조언을 기억하는 것이 중요하다. ‘퍼블릭 API’의 정의는 어느 정도 해석의 여지가 있고, 일부 중요한 행동(예: 부수 효과)은 퍼블릭 API의 범위를 벗어날 수 있다. 하지만 어떤 행동이 중요하고 궁극적으로 신경 써야 하는 것이라면, 퍼블릭 API 여부와 상관없이 테스트되어야 한다.

상대적으로 간단한 클래스는 퍼블릭 API만으로 모든 동작을 쉽게 테스트할 수 있습니다. 이를 통해 코드를 정확하게 검증하고, 구현 세부 사항에 의존하지 않는 더 나은 테스트를 수행할 수 있습니다.

그러나 클래스가 더 복잡해지거나 많은 로직을 포함하게 되면, 퍼블릭 API만으로 모든 동작을 테스트하는 것이 어려워질 수 있습니다. 이런 경우, 코드를 더 작은 단위로 분할하여 테스트할 수 있도록 하는 것이 좋습니다. 이는 코드의 추상화 계층이 너무 큰 상태를 줄이는데 도움이 됩니다.

11.2.3 해결책: 코드를 더 작은 단위로 분할하라

앞서 간단한 논리의 경우 `MortgageAssessor` 클래스에서 퍼블릭 API만으로 충분히 테스트할 수 있었습니다. 예를 들어, 고객의 신용 등급이 좋은지 확인하는 것은 `customer.hasGoodCreditRating()` 함수 호출로 해결되었기 때문에, 퍼블릭 API를 통해 간단히 테스트할 수 있었습니다.

그러나 프라이빗 함수가 복잡한 논리를 포함하게 되면, 이를 테스트하기 위해 프라이빗 함수를 퍼블릭으로 바꾸고 싶어지는 상황이 생길 수 있습니다.

예제 11.9 더 복잡한 신용 등급 확인

```
class MortgageAssessor {  
    private const Double MORTGAGE_MULTIPLIER = 10.0;  
    private const Double GOOD_CREDIT_SCORE_THRESHOLD = 880.0;  
  
    private final CreditScoreService creditScoreService;  
    ...  
  
    MortgageDecision assess(Customer customer) {  
        ...  
    }  
  
    private Result<Boolean, Error> isEligibleForMortgage(  
        Customer customer) {  
        if (customer.hasExistingMortgage() || customer.isBanned()) {  
            return Result.ofValue(false);  
        }  
        return isCreditRatingGood(customer.getId());  
    }  
    /** 테스트를 위해서만 공개 */  
    Result<Boolean, Error> isCreditRatingGood(Int customerId) {  
        CreditScoreResponse response = creditScoreService  
            .query(customerId);  
  
        if (response.errorOccurred()) {  
            return Result.ofError(response.getError());  
        }  
        return Result.ofValue(  
            response.getCreditScore() >= GOOD_CREDIT_SCORE_THRESHOLD);  
    }  
}
```

MortgageAssessor 클래스는 CreditScoreService에 의존한다.

isCreditRatingGood() 함수는 테스트를 위해 공개되었다.

CreditScoreService 서비스에 대해 질의한다.

서비스에 대한 실패나 오류는 리절트 유형으로 전달된다.

점수는 기준값과 비교된다.

고객의 신용 등급을 판단하기 위해 외부 서비스에 의존한다고 가정해 보겠습니다. 이 경우 MortgageAssessor 클래스는 다음과 같은 복잡한 논리를 갖게 됩니다.

- MortgageAssessor 클래스는 CreditScoreService에 의존합니다.
- 고객의 신용 점수를 조회하기 위해 고객 ID로 CreditScoreService에 질의합니다.
- CreditScoreService 호출이 실패할 수 있기 때문에 오류 시나리오를 처리해야 합니다.
- 호출이 성공하면 반환된 점수를 기준 값과 비교하여 고객의 신용 등급이 양호한지 결정합니다.

이러한 복잡성과 오류 시나리오까지 퍼블릭 API만으로 모두 테스트하는 것은 매우 어려워 보입니다. 테스트를 쉽게 하기 위해 isCreditRatingGood() 같은 프라이빗 함수를 퍼블릭으로 만들고 싶어지기 쉬운 상황

입니다. 그러나 이렇게 하면 앞서 살펴본 것과 같은 문제가 발생합니다. 논리가 복잡해짐에 따라 퍼블릭 API만으로 테스트하는 것은 점점 더 어려워집니다.

근본적인 문제: 클래스의 역할이 너무 많다

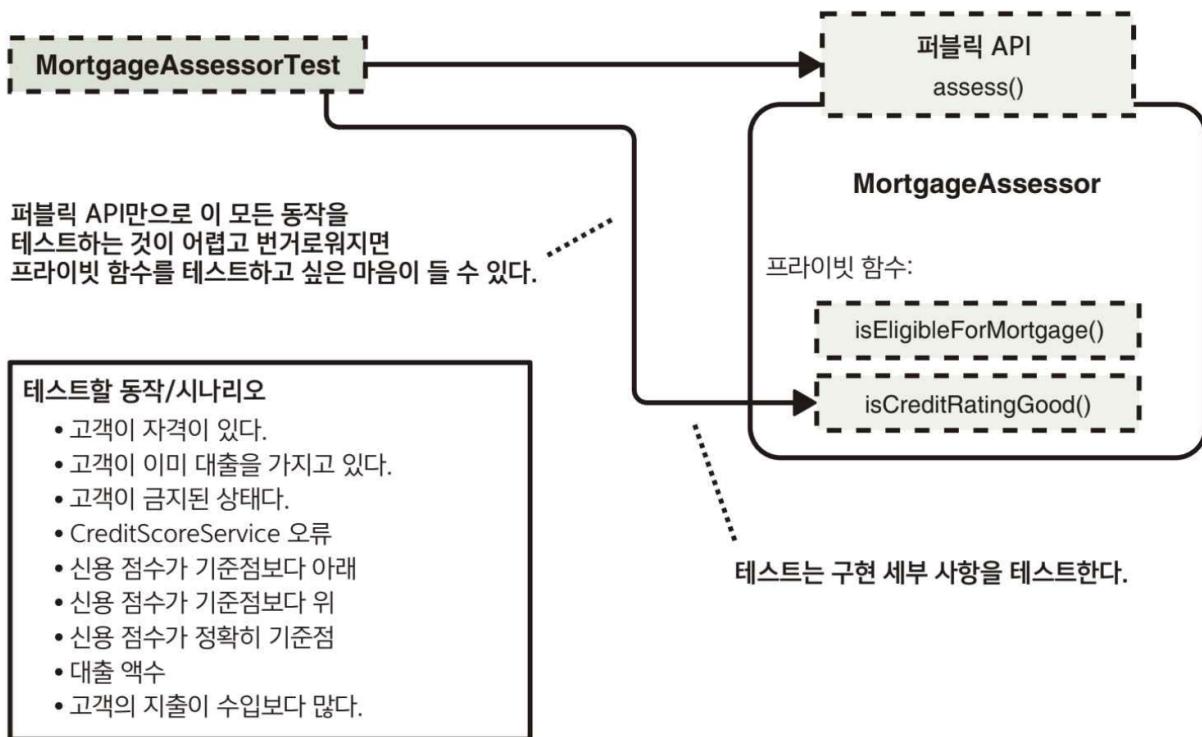


그림 11.1 클래스가 너무 많은 작업을 할 경우 Public API만으로 모든 것을 테스트하기 어려울 수 있다.

2장에서 설명한 것처럼, 하나의 클래스가 너무 많은 개념을 다루지 않도록 하는 것이 중요합니다.

MortgageAssessor 클래스는 다양한 개념을 포함하고 있어, 추상화 계층이 지나치게 비대해졌습니다. 이로 인해 퍼블릭 API만으로 모든 기능을 완벽하게 테스트하기 어려운 상황이 발생한 것입니다. 이러한 경우 코드를 더 작은 단위로 분할하여, 각 기능을 독립적으로 테스트할 수 있도록 하는 것이 바람직합니다.

예제 11.10 두 개의 클래스로 나뉘어진 코드

```
class CreditRatingChecker {           ←  
    private const Double GOOD_CREDIT_SCORE_THRESHOLD = 880.0;          신용 등급이 좋은지 확인하는  
    ...                                                               논리를 구현하는 별도의 클래스  
  
    Result<Boolean, Error> isCreditRatingGood(Int customerId) {  
        CreditScoreService response = creditScoreService  
            .query(customerId);  
        if (response.errorOccurred()) {  
            return Result.ofError(response.getError());  
        }  
        return Result.ofValue(  
            response.getCreditScore() >= GOOD_CREDIT_SCORE_THRESHOLD);  
    }  
  
    class MortgageAssessor {  
        private const Double MORTGAGE_MULTIPLIER = 10.0;  
  
        private final CreditRatingChecker creditRatingChecker;           ←  
        ...  
  
        MortgageDecision assess(Customer customer) {  
            ...  
        }  
  
        private Result<Boolean, Error> isEligibleForMortgage(  
            Customer customer) {  
            if (customer.hasExistingMortgage() || customer.isBanned()) {  
                return Result.ofValue(false);  
            }  
            return creditRatingChecker  
                .isCreditRatingGood(customer.getId());  
        }  
        ...  
    }
```

MortgageAssessor는 CreditRatingChecker에 의존한다.

이 문제를 해결하는 방법은 고객의 신용 등급을 판단하는 논리를 별도의 클래스로 옮기는 것입니다. 예를 들어, CreditRatingChecker라는 클래스를 만들어 신용 등급을 판단하는 역할을 담당하게 할 수 있습니다. 이렇게 하면 MortgageAssessor 클래스는 CreditRatingChecker에 의존하게 되어, 더 이상 복잡한 논리를 포함하지 않으므로 코드가 단순해집니다.

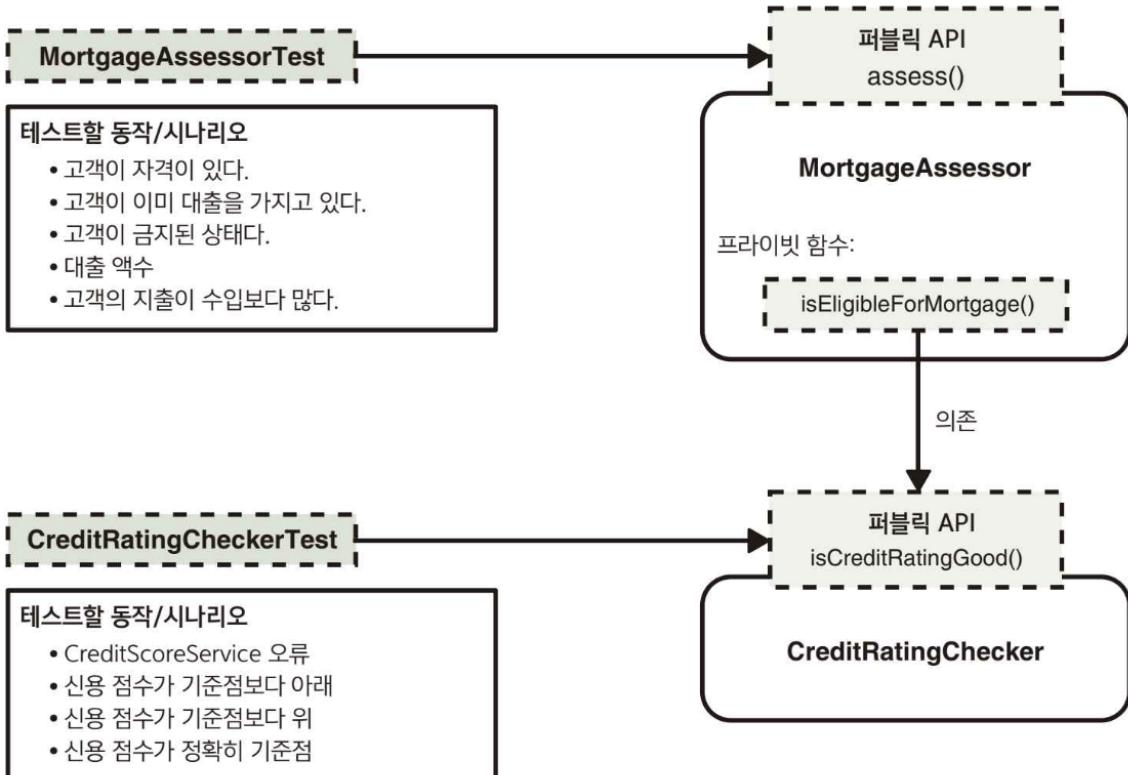


그림 11.2 큰 클래스를 더 작은 클래스로 나누면 코드를 더 쉽게 테스트할 수 있다.

이제 MortgageAssessor와 CreditRatingChecker 클래스는 각자의 역할에 집중하여 더 쉽게 처리할 수 있습니다. 이로써 각각의 퍼블릭 API를 사용해 두 클래스를 독립적으로 테스트할 수 있습니다.

프라이빗 함수를 퍼블릭으로 만들어 테스트하는 것은 중요한 동작을 테스트하지 못하고 있다는 경고 신호로 받아들여야 합니다. 대부분의 경우, 이미 공개된 퍼블릭 함수를 사용해 테스트하는 것이 더 바람직합니다. 만약 이렇게 테스트하기 어렵다면, 클래스(또는 코드 단위)가 너무 크기 때문에 더 작은 클래스나 단위로 분할해야 할 시점임을 의미합니다.

느낀점

이 내용을 통해, 테스트는 구현 세부 사항이 아닌 중요한 동작을 검증하는 데 집중해야 한다는 점과, 하나의 클래스가 너무 많은 책임을 갖지 않도록 역할을 분리하는 것이 중요하다는 점을 깨달았습니다. 복잡한 논리는 별도의 클래스로 추상화하여, 퍼블릭 API를 통해 테스트할 수 있도록 하면 코드의 유지보수성과 테스트 용이성이 크게 향상됩니다. 또한, 추상화 계층이 비대해질 경우 이를 더 작은 단위로 나누어 책임을 명확히 하면, 코드가 더 간결해지고 테스트의 품질도 높아진다는 것을 느꼈습니다.

11.3 한 번에 하나의 동작만 테스트 하라

주어진 코드에 대해 테스트해야 하는 동작이 여러 가지일 때, 보통 각 동작을 테스트하려면 약간씩 다른 시나리오가 필요합니다. 이 경우 각 시나리오와 관련된 동작마다 별도의 테스트 케이스를 작성하는 것이 가장 자연스럽습니다. 하지만 때로는 하나의 시나리오로 여러 동작을 테스트하도록 설정할 수도 있습니다. 다만, 가능하다고 해서 항상 좋은 방법은 아니므로, 각 동작을 독립적으로 테스트할 필요가 있는지 고려하는 것이 중요합니다.

11.3.1 여러 동작을 한꺼번에 테스트하면 테스트가 제대로 안될 수 있다.

예제 11.11 유효한 쿠폰을 추려내기 위한 코드

```
List<Coupon> getValidCoupons(  
    List<Coupon> coupons, Customer customer) {  
    return coupons  
        .filter(coupon -> !coupon.alreadyRedeemed())  
        .filter(coupon -> !coupon.hasExpired())  
        .filter(coupon -> coupon.issuedTo() == customer)  
        .sortBy(coupon -> coupon.getValue(), SortOrder.DESCENDING);  
}
```

`getValidCoupons()` 함수는 주어진 쿠폰 목록에서 유효한 쿠폰만 추려내어 반환하는 역할을 합니다. 이 함수는 여러 동작을 수행하며, 예를 들어 유효한 쿠폰만 반환하고, 이미 사용된 쿠폰이나 유효기간이 지난 쿠폰, 다른 고객에게 발행된 쿠폰은 유효하지 않은 것으로 간주하며, 반환할 쿠폰은 내림차순으로 정렬합니다.

이 함수의 모든 동작을 테스트하려면 각 동작에 대해 별도의 테스트 케이스를 작성하는 것이 가장 명확하고 이해하기 쉽습니다. 하지만 모든 동작을 하나의 테스트 케이스에서 한꺼번에 테스트할 수도 있습니다

예제 11.12 한 번에 모든 것을 테스트하는 코드

```
void testGetValidCoupons_allBehaviors() {  
    Customer customer1 = new Customer("test customer 1");  
    Customer customer2 = new Customer("test customer 2");  
    Coupon redeemed = new Coupon(  
        alreadyRedeemed: true, hasExpired: false,  
        issuedTo: customer1, value: 100);  
    Coupon expired = new Coupon(  
        alreadyRedeemed: false, hasExpired: true,  
        issuedTo: customer1, value: 100);  
    Coupon issuedToSomeoneElse = new Coupon(  
        alreadyRedeemed: false, hasExpired: false,  
        issuedTo: customer2, value: 100);  
    Coupon valid1 = new Coupon(  
        alreadyRedeemed: false, hasExpired: false,  
        issuedTo: customer1, value: 100);  
    Coupon valid2 = new Coupon(  
        alreadyRedeemed: false, hasExpired: false,  
        issuedTo: customer1, value: 150);
```

```
List<Coupon> validCoupons = getValidCoupons(  
    [redeemed, expired, issuedToSomeoneElse, valid1, valid2],  
    customer1);
```

```
assertThat(validCoupons)  
    .containsExactly(valid2, valid1)  
    .inOrder();  
}
```

그러나 이렇게 하면 테스트 코드가 길어지고, `testGetValidCoupons_allBehaviors()` 와 같은 이름으로는 테스트가 정확히 무엇을 하는지 파악하기 어렵습니다. 또한, 모든 동작을 한 번에 테스트하면 코드에 문제가 생겼을 때, 어떤 동작에 문제가 있는지 명확히 알기 어렵습니다.

테스트 케이스가 모든 동작을 테스트하기 때문에 통과하지 못한 경우
테스트 케이스 이름만 봐서는 어떤 동작에 문제가 있는지 알 수 없다.

```
Test case testGetValidCoupons_allBehaviors failed:  
Expected:  
[  
    Coupon(redeemed: false, expired: false,  
            issuedTo: test customer 1, value: 150),  
    Coupon(redeemed: false, expired: false,  
            issuedTo: test customer 1, value: 100)  
]  
But was actually:  
[  
    Coupon(redeemed: false, expired: false,  
            issuedTo: test customer 1, value: 150),  
    Coupon(redeemed: true, expired: false,  
            issuedTo: test customer 1, value: 100),  
    Coupon(redeemed: false, expired: false,  
            issuedTo: test customer 1, value: 100)  
]
```

실패 메시지로부터 어떤 동작에 문제가 있는지
이해하기가 매우 어렵다.

그림 11.3 한 번에 여러 동작을 테스트하면 테스트가 실패하더라도 실패에 대한 설명이 자세히 제공되지 않을 수 있다.

예를 들어, 이미 사용된 쿠폰을 걸러내는 코드에 문제가 생겨도, 단일 테스트 케이스는 단지 실패했을 뿐, 그 원인을 자세히 설명하지 못합니다. 테스트가 이해하기 어렵고 실패 원인이 명확하지 않으면, 다른 개발자의 시간을 낭비하고 버그 발생 가능성을 높일 수 있습니다. 따라서, 테스트는 각 동작별로 분리하여, 실패 시 어떤 동작에 문제가 발생했는지 명확하게 알려줄 수 있도록 하는 것이 바람직합니다.

11.3.2 해결책: 각 동작은 자체 테스트 케이스에서 테스트하라

예제 11.13 한 번에 하나씩 테스트

```
void testGetValidCoupons_validCoupon_included() {  
    Customer customer = new Customer("test customer");  
    Coupon valid = new Coupon(  
        alreadyRedeemed: false, hasExpired: false,  
        issuedTo: customer, value: 100);
```

각 동작은 자신만의
테스트 케이스를 통해 테스트된다.

```
List<Coupon> validCoupons = getValidCoupons([valid], customer);  
  
assertThat(validCoupons).containsExactly(valid);  
}
```

```
void testGetValidCoupons_alreadyRedeemed_excluded() {  
    Customer customer = new Customer("test customer");  
    Coupon redeemed = new Coupon(  
        alreadyRedeemed: true, hasExpired: false,  
        issuedTo: customer, value: 100);
```

각 동작은 자신만의
테스트 케이스를
통해 테스트된다.

```
List<Coupon> validCoupons =  
    getValidCoupons([redeemed], customer);  
  
assertThat(validCoupons).isEmpty();  
}
```

```
void testGetValidCoupons_expired_excluded() { ... }
```

```
void testGetValidCoupons_issuedToDifferentCustomer_excluded() { ... }
```

```
void testGetValidCoupons_returnedInDescendingValueOrder() { ... }
```

더 나은 접근법은 각 동작을 별도의 테스트 케이스로 나누고, 명확하게 이름을 붙이는 것입니다. 예제 11.13에서는 이러한 방식의 테스트 코드가 제시되었으며, 각 테스트 케이스가 간단하고 이해하기 쉬워졌음을 알 수 있습니다. 테스트 케이스 이름을 통해 어떤 동작을 테스트하는지 정확히 알 수 있어, 코드 파악이 수월해지고 테스트 코드의 가독성이 크게 향상되었습니다.

테스트 케이스의 이름은 어떤 동작에 문제가 있는지
즉시 명확하게 알려준다.

```
Test case testGetValidCoupons_alreadyRedeemed_excluded failed:  
Expected:  
[]  
But was actually:  
[  
  Coupon(redeemed: true, expired: false,  
          issuedTo: test customer, value: 100)  
]
```

실패 메시지는 이해하기가 훨씬 더 쉽다.

그림 11.4 한 번에 하나의 동작을 테스트하면 테스트가 통과되지 못한 경우 메시지는 무엇이 문제인지 잘 설명한다.

이렇게 개별 테스트 케이스로 나누면, 테스트 실패 시 어떤 동작에 문제가 있는지 쉽게 파악할 수 있습니다. 예를 들어, 이미 사용된 쿠폰이 제외되는지 확인하는 코드가 잘못되었을 때, `testGetValidCoupons_alreadyRedeemed()` 테스트가 실패하여 어떤 동작에 문제가 있는지 바로 알 수 있습니다. 실패 메시지도 더 명확해져서 문제 해결에 도움이 됩니다.

물론, 각 동작을 독립된 테스트 케이스로 나누면 코드 중복이 생기는 단점이 있지만, 이를 해결하기 위해 **매개변수화된 테스트**를 사용해 반복 코드를 줄일 수 있습니다.

11.3.3 매개변수를 사용한 테스트

일부 테스트 프레임워크는 **매개변수**를 활용하여 여러 시나리오를 테스트할 수 있는 기능을 제공합니다. 이를 통해 테스트 케이스 함수를 한 번 작성하고 매개변수에 다른 값을 설정하여 다양한 시나리오를 테스트할 수 있습니다.

예제 11.14 매개변수를 사용한 테스트

```
[TestCase(true, false, TestName = "이미 사용함")]
[TestCase(false, true, TestName = "유효기간 만료")]
void testGetValidCoupons_excludesInvalidCoupons(
    Boolean alreadyRedeemed, Boolean hasExpired) {
    Customer customer = new Customer("test customer");
    Coupon coupon = new Coupon(
        alreadyRedeemed: alreadyRedeemed,
        hasExpired: hasExpired,
        issuedTo: customer, value: 100);
```

테스트 케이스는 각 매개변수 세트에 대해 한 번씩 실행된다.

테스트 케이스는 함수의 매개변수를 통해 각각 다른 값을 받는다.

매개변수의 값은 테스트 설정 시 사용된다.

```
List<Coupon> validCoupons =
    getValidCoupons([coupon], customer);

assertThat(validCoupons).isEmpty();
}
```

예제 11.14는 `getValidCoupons()` 함수의 두 가지 동작을 매개변수를 통해 테스트하는 방법을 보여줍니다. 이 코드에서는 `testGetValidCoupons_excludesInvalidCoupons()` 함수에 두 개의 불리언 매개변수가 있으며, 각 매개변수에 대해 설정된 값을 통해 여러 테스트 시나리오를 실행합니다.

실패가 잘 설명되도록 하라

예제 11.14에서 각 매개변수 집합에는 관련된 `TestName`이 있다. 테스트가 실패하면 `Test case testGetValidCoupons_excludesInvalidCoupons.alreadyRedeemed failed`와 같은 메시지가 출력되므로 실패에 대해 잘 설명한다(테스트 케이스 이름 뒤에 오류를 초래한 매개변수 세트의 이름 `alreadyRedeemed`가 붙는다).

매개변수를 사용해 테스트 케이스를 작성할 때 각 매개변수 세트에 이름을 추가하는 것은 일반적으로 선택 사항이다. 다만 이를 생략하면 실패에 대한 설명이 제대로 되지 않기 때문에 이름 지정 여부를 결정할 때 테스트 실패 시 메시지가 어떻게 보일지 생각해보는 것이 좋다.

매개변수를 이용한 테스트는 반복적인 코드를 줄이고 다양한 동작을 효율적으로 테스트할 수 있는 좋은 도구입니다. 다만, 테스트 프레임워크마다 매개변수 설정 방식이 다를 수 있습니다. 일부 프레임워크에서는 매개변수 설정이 복잡할 수 있으므로, 사용 중인 언어와 프레임워크에 따라 어떤 방법이 있는지, 그 장단점을 조사해보는 것이 좋습니다.

추가

Java에서는 **JUnit**을 사용해 매개변수화된 테스트를 작성할 수 있습니다. JUnit 5에서는 `@ParameterizedTest`와 `@ValueSource`, `@CsvSource` 등의 어노테이션을 통해 다양한 값으로 테스트

를 반복 실행할 수 있습니다.

<https://tommykim.tistory.com/19>

해당 아티클 참고해보시면 좋을 것 같아요!

11.4 공유 설정을 적절하게 사용하라

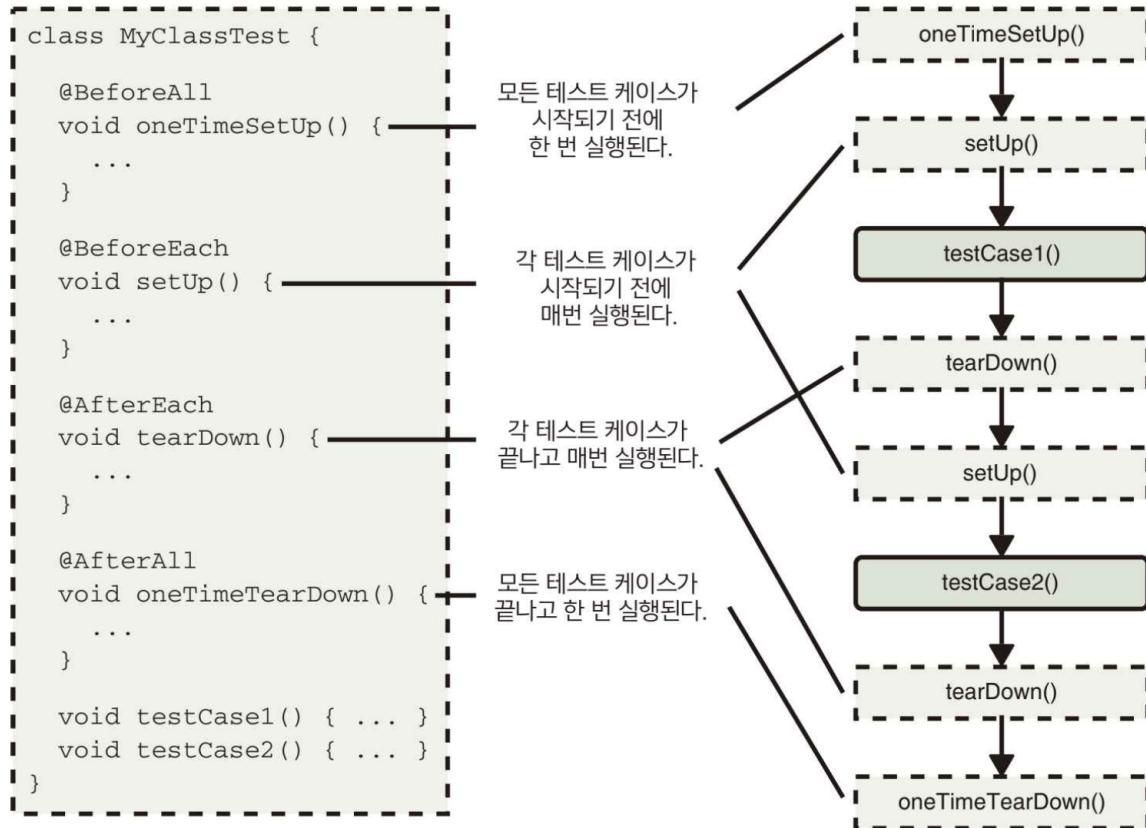


그림 11.5 테스트 프레임워크는 테스트 케이스와 관련하여 다양한 시간에 설정 및 해체 코드를 실행할 수 있는 방법을 제공한다.

테스트 케이스에는 종종 의존성을 설정하거나 테스트 데이터를 준비하는 등 초기화 작업이 필요합니다. 이러한 설정에는 시간과 리소스가 많이 들기 때문에, 많은 테스트 프레임워크는 테스트 케이스 간에 설정을 공유할 수 있는 기능을 제공합니다. 일반적으로 다음 두 시점에서 설정 코드를 실행하도록 구성할 수 있습니다:

1. **BeforeAll:** BeforeAll 블록의 설정 코드는 모든 테스트 케이스 실행 전에 한 번만 실행됩니다. 일부 프레임워크에서는 이를 OneTimeSetup이라고도 합니다.
2. **BeforeEach:** BeforeEach 블록의 설정 코드는 각 테스트 케이스 실행 전에 매번 실행됩니다. 일부 프레임워크에서는 이를 Setup이라고 부릅니다.

이와 유사하게, 설정 후 생성된 상태를 정리하기 위해 해체(tearDown) 코드도 설정할 수 있습니다. 해체 코드는 다음 두 시점에서 실행될 수 있습니다:

1. **AfterAll:** 모든 테스트 케이스가 완료된 후 AfterAll 블록 내의 해체 코드가 한 번 실행됩니다. 일부 프레임워크에서는 이를 OneTimeTearDown이라고도 합니다.

2. **AfterEach**: 각 테스트 케이스가 실행된 후 AfterEach 블록 내의 해체 코드가 매번 실행됩니다. 일부 프레임워크에서는 이를 TearDown이라고 부릅니다.

이러한 설정과 해체 코드를 활용하면 테스트 환경을 효율적으로 관리할 수 있으며, 테스트 실행 순서와 관리 방식에 따라 필요한 준비 작업을 적절히 실행할 수 있습니다.

또한 설정 코드를 공유하면 테스트 케이스 간에 설정을 재사용할 수 있는데, 이는 두 가지 방식으로 이루어질 수 있습니다.

1. **상태 공유 (Sharing State)**: BeforeAll 블록에 설정 코드를 추가하면 모든 테스트 케이스 전에 한 번만 실행되므로, 생성된 모든 상태가 모든 테스트 케이스 간에 공유됩니다. 이 방법은 설정에 시간이 오래 걸리거나 비용이 많이 드는 경우(예: 테스트 서버 시작, 데이터베이스 인스턴스 생성) 유용합니다. 그러나 설정된 상태가 변경 가능할 경우 한 테스트 케이스의 실행 결과가 다른 테스트 케이스에 영향을 줄 위험이 있습니다. 따라서 공유된 상태가 변경되지 않도록 주의해야 합니다.
2. **설정 공유 (Sharing Configuration)**: BeforeEach 블록에 설정 코드를 추가하면 각 테스트 케이스 실행 전에 매번 설정이 초기화됩니다. 즉, 모든 테스트 케이스는 이 설정 코드를 통해 동일한 의존성이나 특정 값을 공유하지만, 테스트 간에 상태가 공유되지는 않습니다. 예를 들어, 모든 테스트 케이스에서 특정 값이나 의존성을 동일하게 설정해야 할 경우, 개별 테스트에서 반복하는 대신 BeforeEach를 통해 설정을 공유하는 것이 유익합니다.

설정 공유는 테스트가 복잡하지 않으면서도 일관성을 유지하도록 돋지만, 잘못 사용하면 테스트의 안정성과 독립성이 떨어질 수 있습니다. 공유된 설정이 테스트의 취약성을 초래하지 않도록 주의해야 하며, 설정 공유가 불필요하게 복잡하거나, 테스트의 독립성을 해치는 상황을 방지해야 합니다.

11.4.1 상태 공유는 문제가 될 수 있다

일반적으로 테스트 케이스는 서로 격리되어야 하며, 한 테스트 케이스에서 발생한 상태 변화가 다른 테스트 케이스에 영향을 주어서는 안 됩니다. 하지만 BeforeAll 블록에서 생성한 공유된 가변 상태는 이러한 규칙을 위반할 수 있습니다.

예제 11.15 데이터베이스에 쓰기 동작을 수행하는 코드

```
class OrderManager {  
    private final Database database;  
  
    void processOrder(Order order) {  
        if (order.containsOutOfStockItem() ||  
            !order.isPaymentComplete()) {  
            database.setOrderStatus(  
                order.getId(), OrderStatus.DELAYED);  
        }  
        .  
    }  
}
```

예제 11.16 테스트 케이스 간 상태 공유

```
class OrderManagerTest {  
  
    private Database database;  
  
    @BeforeAll  
    void oneTimeSetUp() {  
        database = Database.createInstance();  
        database.waitForReady();  
    }  
  
    void testProcessOrder_outOfStockItem_orderDelayed() {  
        Int orderId = 12345;  
        Order order = new Order(  
            orderId: orderId,  
            containsOutOfStockItem: true,  
            isPaymentComplete: true);  
        OrderManager orderManager = new OrderManager(database); ←  
  
        orderManager.processOrder(order); ← 데이터 베이스에서 해당 오더는  
                                         지연 상태로 변경된다.  
        assertThat(database.getOrderStatus(orderId))  
            .isEqualTo(OrderStatus.DELAYED);  
    }  
  
    void testProcessOrder_paymentNotComplete_orderDelayed() {  
        Int orderId = 12345;  
        Order order = new Order(  
            orderId: orderId,  
            containsOutOfStockItem: false,  
            isPaymentComplete: false);  
        OrderManager orderManager = new OrderManager(database); ←  
  
        orderManager.processOrder(order);  
  
        assertThat(database.getOrderStatus(orderId))  
            .isEqualTo(OrderStatus.DELAYED); ← 코드에 문제가 있더라도  
                                         테스트는 통과될 수 있는데,  
                                         이전 테스트 케이스 실행 결과 데이터베이스에  
                                         이미 이 값이 저장되어 있기 때문이다.  
    }  
}
```

예를 들어, OrderManager 클래스가 Database 인스턴스에 의존하고, 데이터베이스 생성에 시간이 많이 걸려 BeforeAll 블록에서 한 번만 생성하여 모든 테스트가 이 인스턴스를 공유하게 된다면 문제가 발생할 수 있습니다.

이 경우, 첫 번째 테스트 케이스인 `testProcessOrder_outOfStockItem_orderDelayed()` 가 실행되어 데이터베이스에 주문 상태를 “DELAYED”로 표시했다고 가정해봅시다. 이어서 두 번째 테스트 케이스 `testProcessOrder_paymentNotComplete_orderDelayed()` 가 실행되면, 첫 번째 테스트의 상태가

데이터베이스에 남아 있어 테스트 결과에 영향을 줄 수 있습니다. 만약 두 번째 테스트에서 코드에 문제가 있더라도, 이전 테스트에 의해 이미 데이터베이스에 저장된 상태 때문에 테스트가 통과할 수 있는 상황이 발생할 수 있습니다.

이처럼 가변 상태를 공유하면 테스트의 정확성을 해칠 수 있으므로 가능하면 상태 공유를 피하는 것이 좋습니다. 만약 상태 공유가 불가피하다면, 한 테스트 케이스에서 변경된 상태가 다른 테스트 케이스에 영향을 미치지 않도록 신중하게 관리해야 합니다.

11.4.2 해결책: 상태를 공유하지 않거나 초기화해라

가변 상태를 공유할 때 발생하는 문제를 해결하는 가장 확실한 방법은 애초에 상태를 공유하지 않는 것입니다. 예를 들어, OrderManagerTest에서 테스트 케이스 간에 Database 인스턴스를 공유하지 않도록 할 수 있다면, 각 테스트 케이스 내에서 또는 BeforeEach 블록을 사용해 각 테스트마다 새로운 인스턴스를 생성하는 방법을 고려할 수 있습니다.

또 다른 해결책으로는 테스트 더블을 사용하는 것입니다. Database 클래스의 대안으로, 빠르게 인스턴스를 생성할 수 있는 FakeDatabase를 사용하면, 각 테스트 케이스마다 새로운 인스턴스를 생성해 상태가 공유되지 않도록 할 수 있습니다.

예제 11.17 테스트 케이스 간 상태 초기화

```
class OrderManagerTest {  
  
    private Database database;  
  
    @BeforeAll  
    void oneTimeSetUp() {  
        database = Database.createInstance();  
        database.waitForReady();  
    }  
  
    @AfterEach  
    void tearDown() {  
        database.reset();  
    }  
}
```

데이터베이스는 각 테스트 케이스가 실행된 후에 매번 초기화된다.

만약 데이터베이스 생성이 너무 느리거나 비용이 많이 들고, FakeDatabase 같은 대체가 없다면 상태 공유가 불가피해집니다. 이 경우, AfterEach 블록을 사용해 각 테스트 케이스가 끝난 후 상태를 초기화해야 합니다. 이렇게 하면 다음 테스트가 실행되기 전에 반드시 초기 상태가 보장됩니다.

NOTE 전역 상태

테스트 케이스 간 상태 공유가 테스트 코드를 통해서만 되는 것은 아니라는 점에 유의할 필요가 있다. 테스트 대상 코드가 전역 상태를 유지한다면 테스트 케이스마다 이 전역 상태를 확실하게 초기화해야 한다. 9장에서 전역 상태에 대해 논의할 때, 전역 상태는 갖지 않는 것이 일반적으로 최선이라고 결론 내렸다. 전역 상태가 코드의 테스트 용이성에 영향을 미친다는 점도 전역 상태를 사용하지 말아야 하는 이유 중 하나다.

일반적으로 가변적인 상태는 테스트 케이스 간에 공유하지 않는 것이 가장 바람직합니다. 하지만 상태 공유가 꼭 필요한 상황이라면, 각 테스트 간에 상태가 초기화되도록 주의해 한 테스트의 결과가 다른 테스트에 영향을 주지 않도록 해야 합니다.

11.4.3 설정 공유는 문제가 될 수 있다

테스트 케이스 간에 설정을 공유하는 것은 코드 중복을 줄이는 데 도움이 되지만, 이는 테스트의 신뢰성을 해칠 수 있습니다.

예제 11.18 우편 라벨 코드

```
class OrderPostageManager {  
    ...  
  
    PostageLabel getPostageLabel(Order order) {  
        return new PostageLabel(  
            address: order.getCustomer().getAddress(),  
            isLargePackage: order.getItems().size() > 2,  
        );  
    }  
}
```

주문 항목이 세 개 이상이면
대형 배송 박스로 표시된다.

예제 11.19 테스트 설정 공유

```
class OrderPostageManagerTest {  
    private Order testOrder;  
  
    @BeforeEach  
    void setUp() {  
        testOrder = new Order(  
            customer: new Customer(  
                address: new Address("Test address"),  
            ),  
            items: [  
                new Item(name: "Test item 1"),  
                new Item(name: "Test item 2"),  
                new Item(name: "Test item 3"),  
            ]);  
    }  
    ...  
}
```

공유되는 설정

```
void testGetPostageLabel_threeItems_largePackage() {  
    PostageManager postageManager = new PostageManager();  
  
    PostageLabel label =  
        postageManager.getPostageLabel(testOrder);  
  
    assertThat(label.isLargePackage()).isTrue();  
}  
  
...  
}
```

테스트 케이스는 공유된 설정에서 주문에 추가되는 항목이 정확히 세 개라는 사실에 의존한다.

예를 들어, Order 클래스의 인스턴스를 생성하는 데 여러 클래스의 인스턴스가 필요하다면 BeforeEach 블록에서 Order 인스턴스를 미리 생성하여 반복을 피할 수 있습니다. 그러나 이 방식은 설정의 특성에 테스트가 의존하게 되는 문제를 일으킬 수 있습니다.

예제 11.20 새로운 기능

```
class PostageManager {  
    ...  
  
    PostageLabel getPostageLabel(Order order) {  
        return new PostageLabel(  
            address: order.getCustomer().getAddress(),  
            isLargePackage: order.getItems().size() > 2,  
            isHazardous: containsHazardousItem(order.getItems()));  
    }  
  
    private static Boolean containsHazardousItem(List<Item> items) {  
        return items.anyMatch(item -> item.isHazardous());  
    }  
}
```

배송 박스가 위험물을 포함하고 있는지 표시하는 새로운 기능

예제 11.21 공유 설정의 잘못된 변경

```
class OrderPostageManagerTest {  
    private Order testOrder;  
  
    @BeforeEach  
    void setUp() {  
        testOrder = new Order(  
            customer: new Customer(  
                address: new Address("Test address"),  
            ),  
            items: [  
                new Item(name: "Test item 1"),  
                new Item(name: "Test item 2"),  
                new Item(name: "Test item 3"),  
                new Item(name: "Hazardous item", isHazardous: true),  
            ]);  
    }  
    ...  
  
    void testGetPostageLabel_threeItems_largePackage() { ... }  
  
    void testGetPostageLabel_hazardousItem_isHazardous() {  
        PostageManager postageManager = new PostageManager();  
  
        PostageLabel label =  
            postageManager.getPostageLabel(testOrder);  
  
        assertThat(label.isHazardous()).isTrue();  
    }  
    ...  
}
```

공유된 설정에서 네 번째 항목이 주문에 추가된다.

세 개의 주문 항목에 대해 의도된 테스트가 네 개의 항목에 대해 테스트한다.

라벨이 위험 품목이라고 표시되는지를 확인하기 위한 새로운 테스트 케이스

예를 들어, 한 개발자가 `getPostageLabel()` 함수에 새 기능을 추가하여 주문 항목 중 위험 물품이 있을 경우 이를 표시하는 로직을 추가했다고 가정해봅시다. 이를 테스트하기 위해, 기존 `Order` 객체에 위험 항목을 추가해 새로운 테스트 케이스를 작성할 수 있다고 생각할 수 있습니다.

하지만 이로 인해 기존 테스트 케이스, 예를 들어 `testGetPostageLabel_threeItems_largePackage()`가 의도했던 동작을 더 이상 정확히 테스트하지 못하게 될 위험이 있습니다. 이 테스트는 정확히 세 개의 항목이 있을 때의 동작을 확인하려 했지만, 이제 네 개의 항목으로 실행되기 때문입니다.

테스트 상수 공유

테스트 설정을 공유하는 방법으로 `BeforeEach`와 `BeforeAll` 블록만 있는 것은 아니다. 공유 테스트 상수를 사용하면 정확히 동일한 결과를 얻을 수 있지만, 앞에서 논의한 문제는 여전히 발생할 가능성이 있다. `Order PostageManagerTest`가 테스트 오더를 `BeforeEach` 블록 대신 공유 상수로 설정한다면 다음 코드에서와 같다.

```
class OrderPostageManagerTest {
    private const Order TEST_ORDER = new Order(
        customer: new Customer(
            address: new Address("Test address"),
        ),
        items: [
            new Item(name: "Test item 1"),
            new Item(name: "Test item 2"),
            new Item(name: "Test item 3"),
            new Item(name: "Hazardous item", isHazardous: true),
        ],
    );
    ...
}
```

공유 테스트 상수

기술적으로 이 방법은 테스트 케이스 간에 상태를 공유하지만 불변 데이터 유형을 사용해 상수를 생성하는 것이 바람직하고 이것은 가변적인 상태는 공유되지 않아야 한다는 것을 의미한다. 앞서 예로 든 코드에서 `Order` 클래스는 불변이다. 불변이 아니라면, `Order` 인스턴스를 공유 상수를 통해 공유하는 것은 11.4.1절에서 논의한 이유로 인해 훨씬 더 바람직하지 않다.

이처럼, 테스트 케이스 간에 설정을 공유하면 코드 중복을 줄이는 데는 도움이 될 수 있지만, 각 테스트가 특정 조건이나 상태에 의존할 경우 오히려 신뢰성을 해칠 수 있습니다.

11.4.4 해결책: 중요한 설정은 테스트 케이스 내에서 정의하라

예제 11.22 테스트 케이스 내에서 이루어지는 중요한 설정

```
class OrderPostageManagerTest {  
    ...  
  
    void testGetPostageLabel_threeItems_largePackage() {  
        Order order = createOrderWithItems([  
            new Item(name: "Test item 1"),  
            new Item(name: "Test item 2"),  
            new Item(name: "Test item 3"),  
        ]);  
        PostageManager postageManager = new PostageManager();  
  
        PostageLabel label = postageManager.getPostageLabel(order);  
  
        assertThat(label.isLargePackage()).isTrue();  
    }  
  
    void testGetPostageLabel_hazardousItem_isHazardous() {  
        Order order = createOrderWithItems([  
            new Item(name: "Hazardous item", isHazardous: true),  
        ]);  
        PostageManager postageManager = new PostageManager();  
  
        PostageLabel label = postageManager.getPostageLabel(order);  
  
        assertThat(label.isHazardous()).isTrue();  
    }  
    ...  
  
    private static Order createOrderWithItems(List<Item> items) {  
        return new Order(  
            customer: new Customer(  
                address: new Address("Test address"),  
            ),  
            items: items);  
    }  
}
```

중요한 사항은
테스트 케이스 내에서
설정한다.

특정 항목으로
주문을 생성하기 위한
헬퍼 함수

중요한 설정이 특정 테스트 케이스에 직접적으로 영향을 미친다면, 그 설정을 테스트 케이스 내에서 정의하는 것이 가장 안전합니다. 반복되는 설정을 매번 테스트 케이스 안에서 직접 작성하는 것이 번거로울 수 있지만, 보통 헬퍼 함수를 사용해 이 작업을 더 쉽게 할 수 있습니다.

예를 들어, `getPostageLabel()` 함수를 테스트하기 위해 `Order` 인스턴스를 만드는 작업이 복잡하더라도, 헬퍼 함수를 사용해 이를 간편하게 처리할 수 있습니다. 각 테스트 케이스는 이 헬퍼 함수를 호출하여 필요한 값을 설정하므로, 설정을 공유하지 않고도 코드 중복을 줄이면서 설정 공유로 인한 문제를 피할 수 있습니다.

11.4.5 설정 공유가 적절한 경우

테스트 설정을 무조건 공유하지 말아야 한다는 것은 아닙니다. 테스트 결과에 직접 영향을 미치지 않는 설정이라면, 설정을 공유하여 코드의 반복을 줄이고 테스트를 더 간결하게 만들 수 있습니다.

예제 11.23 공유 구성의 적절한 사용

```
class OrderPostageManagerTest {  
    private const OrderMetadata ORDER_METADATA =  
        new OrderMetadata(  
            timestamp: Instant.ofEpochSecond(0),  
            serverIp: new IPAddress(0, 0, 0, 0));  
  
    void testGetPostageLabel_threeItems_largePackage() { ... }  
    void testGetPostageLabel_hazardousItem_isHazardous() { ... }  
    ...  
  
    void testGetPostageLabel_containsCustomerAddress() {  
        Address address = new Address("Test customer address");  
        Order order = new Order(  
            metadata: ORDER_METADATA,  
            customer: new Customer(  
                address: address,  
            ),  
        items: []);  
  
        PostageLabel label = postageManager.getPostageLabel(order);  
  
        assertThat(label.getAddress()).isEqualTo(address);  
    }  
    ...  
}
```

OrderMetaData 인스턴스가
공유 설정을 통해 생성된다.

공유된 OrderMetadata는
테스트 케이스에서 사용된다.

```
private static Order createOrderWithItems(List<Item> items) {  
    return new Order(  
        metadata: ORDER_METADATA,  
        customer: new Customer(  
            address: new Address("Test address"),  
        ),  
        items: items);  
}  
}
```

공유된 OrderMetadata는
테스트 케이스에서 사용된다.

예를 들어, Order 클래스의 인스턴스를 생성하려면 일부 메타데이터가 필요하지만, 이 메타데이터가 테스트 결과에 영향을 미치지 않는다고 가정해봅시다. 이 경우 메타데이터를 모든 테스트 케이스에서 반복 설정하지 않고, 공유 설정으로 정의하는 것이 효율적입니다.

함수 매개변수는 꼭 필요한 것만 갖는 것이 이상적이다

9장에서는 함수 매개변수가 뚜렷한 목적을 가질 수 있는 방법에 대해 살펴봤는데, 이 말은 함수가 필요한 값만 받는다는 것을 의미한다. 코드에 대한 테스트를 할 때 필요는 하지만 코드의 동작에는 별로 관련이 없는 설정을 해야 한다면, 함수(또는 생성자) 매개변수의 목적이 명확하지 않다는 것을 알려주는 신호일 수도 있다. 예를 들어 `PostageManager.getPostageLabel()` 함수가 `Order` 클래스의 전체 인스턴스를 받는 대신 `Address` 인스턴스와 주문 항목 목록을 사용해야 한다고 주장할 수 있다. 이 경우에는 테스트에서 `OrderMetadata` 인스턴스와 같은 관련 없는 항목을 생성할 필요가 없다.

설정 공유는 반복 작업을 줄이고 비용이 많이 드는 설정을 효율적으로 관리하는 데 유용하지만, 잘못 사용하면 테스트의 명확성과 효과를 떨어뜨릴 수 있습니다. 설정을 공유할 때는 신중하게 판단하여 사용하는 것이 중요합니다.

느낀점

상태 공유는 데이터 자원이 그대로 유지되기 때문에, 정말로 해당 자원이 공유가 되어야 하는지를 먼저 생각을 해보고 도입을 해야겠다는 생각이 들었습니다. 설정 공유는 데이터 자원이 각 테스트 케이스마다 초기화 되기 때문에 위험성은 상태공유 보다는 낮으나, 웬만하면 헬퍼 함수를 사용해서 테스트 케이스 내에서 정의를 하도록 해야겠다는 생각이 들었습니다!

11.5 적절한 어시션 확인자를 사용하라

적절한 어서션 매처를 사용하는 것은 테스트의 결과를 명확하게 검증하고, 실패 시 이유를 잘 설명하기 위해 중요합니다.

```
assertThat(someValue).isEqualTo("expected value");
assertThat(someList).contains("expected value");
```

위와 같은 어서션은 테스트 실패 시 이유를 잘 설명하는 메시지를 제공합니다. 올바른 어서션 매처를 선택하면 실패 원인을 명확하게 알려줘 디버깅이 쉬워지며, 이는 좋은 단위 테스트의 핵심 요소입니다.

11.5.1 부적합한 확인자는 테스트 실패를 잘 설명하지 못할 수 있다.

부적절한 매처를 사용할 경우, 테스트가 원래 의도보다 더 많은 요소를 검증하게 되거나, 필요하지 않은 순서를 검증해 불필요하게 실패할 수 있습니다.

예제 11.24 TextWidget 코드

```
class TextWidget {  
    private const ImmutableList<String> STANDARD_CLASS_NAMES =  
        ["text-widget", "selectable"];  
    private final ImmutableList<String> customClassNames;  
  
    TextWidget(List<String> customClassNames) {  
        this.customClassNames = ImmutableList.copyOf(customClassNames);  
    }  
  
    /**  
     * 구성요소에 대한 클래스 이름을 반환한다. 반환된 리스트에서 클래스 이름은  
     * 특정한 순서를 갖지 않는다.  
     */  
    ImmutableList<String> getclassNames() {  
        return STANDARD_CLASS_NAMES.concat(customClassNames);  
    }  
}
```

하드 코딩된
CSS 클래스 이름

생성자를 통해 제공되는
사용자 지정 클래스 이름

모든 클래스 이름을
반환한다.

예제 11.25 과도하게 제한된 테스트 어서션

```
void testGetclassNames_containsCustomclassNames() {  
    TextWidget textWidget = new TextWidget(  
        ["custom_class_1", "custom_class_2"]);  
  
    assertThat(textWidget.getclassNames()).isEqualTo([  
        "text-widget",  
        "selectable",  
        "custom_class_1",  
        "custom_class_2",  
    ]);  
}
```

예를 들어, 클래스 이름 목록에 특정 사용자 정의 클래스가 포함되어 있는지를 확인하기 위해 `isEqualTo`를 사용하면, 순서가 중요한 상황이 아니더라도 불필요하게 순서까지 확인하게 되어 잘못된 실패를 일으킬 수 있습니다. 대신 `contains` 같은 어서션 매처를 사용하여 원하는 값만 포함되었는지 확인하면 문제를 해결할 수 있습니다.

예제 11.26 실패 이유를 제대로 설명해주지 않는 테스트 어서션

```
void testGetClassNames_containsCustomclassNames() {  
    TextWidget textWidget = new TextWidget(  
        ["custom_class_1", "custom_class_2"]);  
  
    ImmutableList<String> result = textWidget.getClassNames();  
  
    assertThat(result.contains("custom_class_1")).isTrue();  
    assertThat(result.contains("custom_class_2")).isTrue();  
}
```

테스트 케이스 `testGetClassNames_containsCustomclassNames` 실패:
예상한 값은 참이지만 테스트 결과는 거짓입니다.

실패 메시지는 문제를 거의 설명하지 못한다.

그림 11.6 부적합한 어서션 확인자를 사용하면 시험 실패 시 실패의 이유가 제대로 설명되지 않을 수 있다.

다만, `contains`를 사용할 때도 테스트가 실패했을 때 왜 실패했는지를 제대로 설명하지 못하는 경우가 있습니다. 이런 경우 적절한 매커니즘 선택하여, 단순히 테스트가 통과하지 않는 상황을 넘어, 구체적으로 어떤 차이로 인해 실패했는지를 명확하게 알려주는 어서션을 사용하는 것이 중요합니다.

11.5.2 해결책 적절한 확인자를 사용하라

적절한 어서션 매커니즘을 사용하는 것은 테스트의 신뢰성과 가독성을 높이는 중요한 방법입니다. 최신 테스트 어서션 도구들은 다양한 매커니즘을 제공해, 리스트에서 특정 항목이 포함되었는지 순서와 관계없이 검증하는 등의 기능을 제공합니다.

예제 11.27 적절한 어서션 확인자

```
testGetClassNames_containsCustomClassNames() {  
    TextWidget textWidget = new TextWidget(  
        ["custom_class_1", "custom_class_2"]);  
  
    assertThat(textWidget.getClassNames())  
        .containsAtLeast ("custom_class_1", "custom_class_2");  
}
```

테스트 케이스 `testGetClassNames_containsCustomClassNames` 실패:

Not true that
[**text-widget, selectable, custom_class_2**]는
contains at least
[**custom_class_1, custom_class_2**]를 포함하지 않습니다.

빠진 항목: **custom_class_1**

실패 메시지는 실제 동작과 예상 동작이
어떻게 다른지 명확하게 설명한다.

그림 11.7 적절한 어서션 확인자를 사용하면 테스트가 실패할 경우 실패의 이유를 잘 설명한다.

예를 들어, 자바의 Truth 라이브러리에는 `containsAtLeast()` 매처를 제공합니다. 이러한 매처를 사용하면 순서나 일부 항목 변화에 의한 불필요한 실패를 방지하면서도 원하는 항목이 포함되었는지 확인할 수 있습니다.

`containsAtLeast()`를 사용한 테스트는 클래스 이름의 순서가 바뀌거나 일부 항목이 변경되어도 실패하지 않으면서, 원하는 값이 포함되지 않은 경우에만 실패하도록 합니다. 이로 인해 테스트가 실패했을 때 실패 원인을 명확하게 설명하는 메시지를 제공해, 문제를 빠르게 이해하고 수정할 수 있도록 돕습니다.

따라서 테스트에서 적절한 매처를 선택하는 것은 단순히 테스트가 통과하는지 여부를 넘어, 테스트 실패 시 원인을 잘 설명하고 쉽게 이해할 수 있게 만드는 데 필수적입니다.

느낀점

이 내용을 통해 적절한 어서션 매처 선택이 테스트의 신뢰성과 가독성에 얼마나 중요한지 알게 되었습니다. 테스트가 단순히 통과하는지 확인하는 것뿐 아니라, 실패했을 때 구체적으로 어떤 부분이 잘못되었는지 명확하게 알려주는 것이 더욱 중요하다는 점을 배웠습니다!!

책이 출판되고 꽤 오랜시간이 흘렀는데, 한번 Truth 말고도 더 좋은 라이브러리를 찾아보아야 겠다는 생각이 들었습니다!

11.6 테스트 용이성을 위해 의존성을 주입하라

의존성 주입(DI)을 사용하면 코드의 유연성과 테스트 용이성이 크게 향상됩니다.

이전 장에서 설명한 것처럼, 테스트는 대상 코드가 의존하는 객체와 상호작용하는 방식이나 부수 효과를 확인해야 하는 경우가 많습니다.

또한, 의존성을 대체하기 위해 테스트 더블(예: Mock 객체)을 사용하는 방법도 살펴봤습니다. 의존성 주입을 통해, 테스트 코드가 특정 의존성을 직접 제공할 수 있게 되면 필요한 값이나 설정을 주입하여 다양한 테스트 시나리오를 구성할 수 있습니다. 따라서, 의존성 주입을 적용하면 코드의 테스트 가능성이 높아지고, 필요한 동작을 보다 쉽게 검증할 수 있습니다.

11.6.1 하드 코딩 된 의존성을 테스트를 불가능하게 할 수 있다.

예제 11.28 의존성 주입을 하지 않는 클래스

```
class InvoiceReminder {  
    private final AddressBook addressBook;  
    private final EmailSender emailSender;  
  
    InvoiceReminder() {  
        this.addressBook = DataStore.getAddressBook();  
        this.emailSender = new EmailSenderImpl();  
    }  
  
    @CheckReturnValue  
    Boolean sendReminder(Invoice invoice) {  
        EmailAddress? address =  
            addressBook.lookupEmailAddress(invoice.getCustomerId());  
        if (address == null) {  
            return false;  
        }  
        return emailSender.send(  
            address,  
            InvoiceReminderTemplate.generate(invoice));  
    }  
}
```

의존 객체가 생성자에서 생성된다.

addressBook을 사용해 이메일 주소를 조회한다.

emailSender를 사용해 이메일을 전송한다.

InvoiceReminder 클래스는 의존성 주입을 사용하지 않고 생성자에서 직접 AddressBook과 EmailSender 인스턴스를 생성합니다. 이 방식은 다음과 같은 문제를 야기하여 테스트를 어렵게 만듭니다.

- 데이터베이스 연결 문제: InvoiceReminder는 Datastore.getAddressBook()을 호출하여 AddressBook 인스턴스를 생성하는데, 이 인스턴스는 실제 고객 데이터베이스에 연결되어 고객의 연락처 정보를 조회합니다. 실 데이터가 변경될 수 있기 때문에 테스트에 적합하지 않으며, 테스트 환경에서 데이터베이스 접근 권한이 없을 수도 있습니다.

2. 외부 세계와의 부수 효과: InvoiceReminder는 직접 EmailSenderImpl을 생성하여 실제 이메일을 발송합니다. 이는 테스트에서 피해야 하는 부수 효과로, 외부 세계에 영향을 미칠 수 있습니다.

이 문제를 해결하려면 AddressBook과 EmailSender에 대한 테스트 더블(Mock 객체)을 사용하는 것이 이상적입니다. 그러나 현재 InvoiceReminder는 의존성 주입을 지원하지 않아 이를 구현할 수 없고, 결과적으로 테스트가 어려워져 버그 발생 가능성이 높아집니다.

11.6.2 해결책: 의존성 주입을 사용하라

예제 11.29 의존성 주입을 사용한 클래스

```
class InvoiceReminder {  
    private final AddressBook addressBook;  
    private final EmailSender emailSender;  
  
    InvoiceReminder(  
        AddressBook addressBook,  
        EmailSender emailSender) {  
        this.addressBook = addressBook;  
        this.emailSender = emailSender;  
    }  
}
```

생성자를 통해
주입되는 의존성

```
static InvoiceReminder create() {  
    return new InvoiceReminder(  
        DataStore.getAddressBook(),  
        new EmailSenderImpl());  
}
```

정적 팩토리 함수

```
@CheckReturnValue  
Boolean sendReminder(Invoice invoice) {  
    EmailAddress? address =  
        addressBook.lookupEmailAddress(invoice.getCustomerId());  
    if (address == null) {  
        return false;  
    }  
    return emailSender.send(  
        address,  
        InvoiceReminderTemplate.generate(invoice));  
}  
}
```

```
...  
FakeAddressBook addressBook = new FakeAddressBook();  
fakeAddressBook.addEntry(  
    customerId: 123456,  
    emailAddress: "test@example.com");  
FakeEmailSender emailSender = new FakeEmailSender();
```

```
InvoiceReminder invoiceReminder =  
    new InvoiceReminder(addressBook, emailSender);
```

의존성 주입(DI)을 사용하면 InvoiceReminder 클래스의 테스트가 훨씬 쉬워지고, 직접 객체를 생성하는 방식에서 발생했던 문제를 해결할 수 있습니다. DI를 통해 InvoiceReminder는 생성자에서 외부로부터

AddressBook과 EmailSender 객체를 주입받도록 수정됩니다. 이로써, 테스트 환경에서는 FakeAddressBook과 FakeEmailSender 같은 테스트 더블(Mock 객체)을 쉽게 주입해 테스트를 수행할 수 있습니다.

모듈화와 느슨한 결합이 이루어지면 코드 재사용이 용이해지고 테스트가 쉬워지는 경향이 있습니다. 의존성 주입은 코드 모듈화를 돋고, 코드의 테스트 용이성을 높이는 데 효과적인 방법입니다.

느낀점

이 내용을 통해 의존성 주입(DI)이 테스트 용이성에 얼마나 중요한 역할을 하는지 알게 되었습니다. 코드에서 필요한 객체를 직접 생성하지 않고 외부에서 주입받도록 하면, 테스트 환경에서 실제 의존성을 테스트 더블로 대체할 수 있어 다양한 시나리오를 검증하기가 훨씬 쉬워지므로, DI를 잘 활용하는게 중요할 것 같습니다!!

11.7 테스트에 대한 몇 가지 결론

소프트웨어 테스트는 다양한 유형과 수준이 있으며, 이 장에서는 주로 단위 테스트에 대해 다루었습니다. 개발자가 자주 접하는 테스트 수준으로는 단위 테스트 외에 통합 테스트와 종단 간 테스트가 있습니다. 통합 테스트는 여러 모듈과 하위 시스템의 연계를 확인하며, 종단 간 테스트(E2E)는 시스템 전체의 흐름을 테스트합니다.

또한, 몇 가지 유용한 테스트 개념이 있습니다. 회귀 테스트는 소프트웨어 기능이 의도치 않게 변하지 않았음을 확인하고, 골든 테스트는 특정 입력에 대해 예측된 출력을 스냅샷으로 저장해 변화를 감지합니다. 퍼즈 테스트는 무작위 값으로 시스템이 정상적으로 작동하는지를 확인하는 방법입니다.

소프트웨어 품질을 높이기 위해서는 이러한 다양한 테스트 기술을 적절히 조합하고, 새로운 툴과 기술에 대한 최신 정보를 유지하는 것이 중요합니다.

결국 이 장에서 말하는 결론은 다양한 테스트 기법을 적절하게 활용하여 소프트웨어의 신뢰성을 높이는 것이 중요하다는 점입니다. 단위 테스트만으로는 충분하지 않으므로, 통합 테스트와 종단 간 테스트 같은 다른 수준의 테스트도 함께 수행해야 하며, 새로운 도구와 기술을 계속해서 습득하는 것이 좋다고 합니다. 또한, 적절한 테스트 설계와 기법을 통해 코드의 유지보수성과 품질을 높일 수 있다는 점을 강조하고 있습니다.

요약

1. 중요한 동작을 테스트: 개별 함수보다 실제로 중요한 동작을 파악하고 테스트 케이스를 작성하는 것이 효과적입니다. 프라이빗 함수를 테스트하기보다는 중요한 결과를 검증하는 것이 더 중요합니다.
2. 하나의 동작씩 테스트: 한 번에 한 가지 동작만 테스트하면, 테스트가 실패했을 때 원인을 명확히 파악하기 쉬우며 코드 이해도도 높아집니다.
3. 설정 공유의 장단점: 테스트 설정을 공유하면 반복 작업을 줄일 수 있지만, 잘못 사용할 경우 신뢰성이 떨어질 수 있으므로 주의가 필요합니다.
4. 의존성 주입 사용: 의존성 주입을 활용하면 코드의 테스트 용이성이 크게 향상됩니다.
5. 다양한 테스트 활용: 단위 테스트는 기본이지만, 고품질 소프트웨어를 위해서는 통합 테스트, 종단 간 테

스트 등 다양한 테스트 기법을 함께 사용하는 것이 필요합니다.