

Chap10. 단위 테스트의 원칙

이번 장의 목표

- 단위 테스트의 기본사항
- 좋은 단위 테스트가 되기 위한 조건
- 테스트 더블
- 테스트 철학

들어가며

개발자들은 대체로 단위 테스트에 집중하여, 실제 코드가 잘 작동하는지를 확인하기 위해 테스트 코드를 작성합니다. 단위 테스트란 비교적 격리된 방식으로 코드의 독립적인 단위를 검증하는 것인데, 이 “단위”가 무엇인지는 상황에 따라 다를 수 있지만, 일반적으로 클래스, 함수, 혹은 코드 파일을 의미하는 경우가 많습니다.

테스트의 격리 수준에 대한 관점도 다양하여, 일부 개발자는 테스트 대상 코드가 의존하는 코드를 차단하는 방식을 선호하는 반면, 다른 개발자는 이를 포함하여 테스트하는 방식을 선호합니다. 단위 테스트의 정확한 정의가 없는 상황에서도, 코드를 철저히 테스트하고 유지보수 가능한 방식으로 관리하는 것이 중요합니다.

10.1 단위 테스트의 기초

이 장에서는 단위 테스트와 관련된 주요 개념과 용어를 설명합니다.

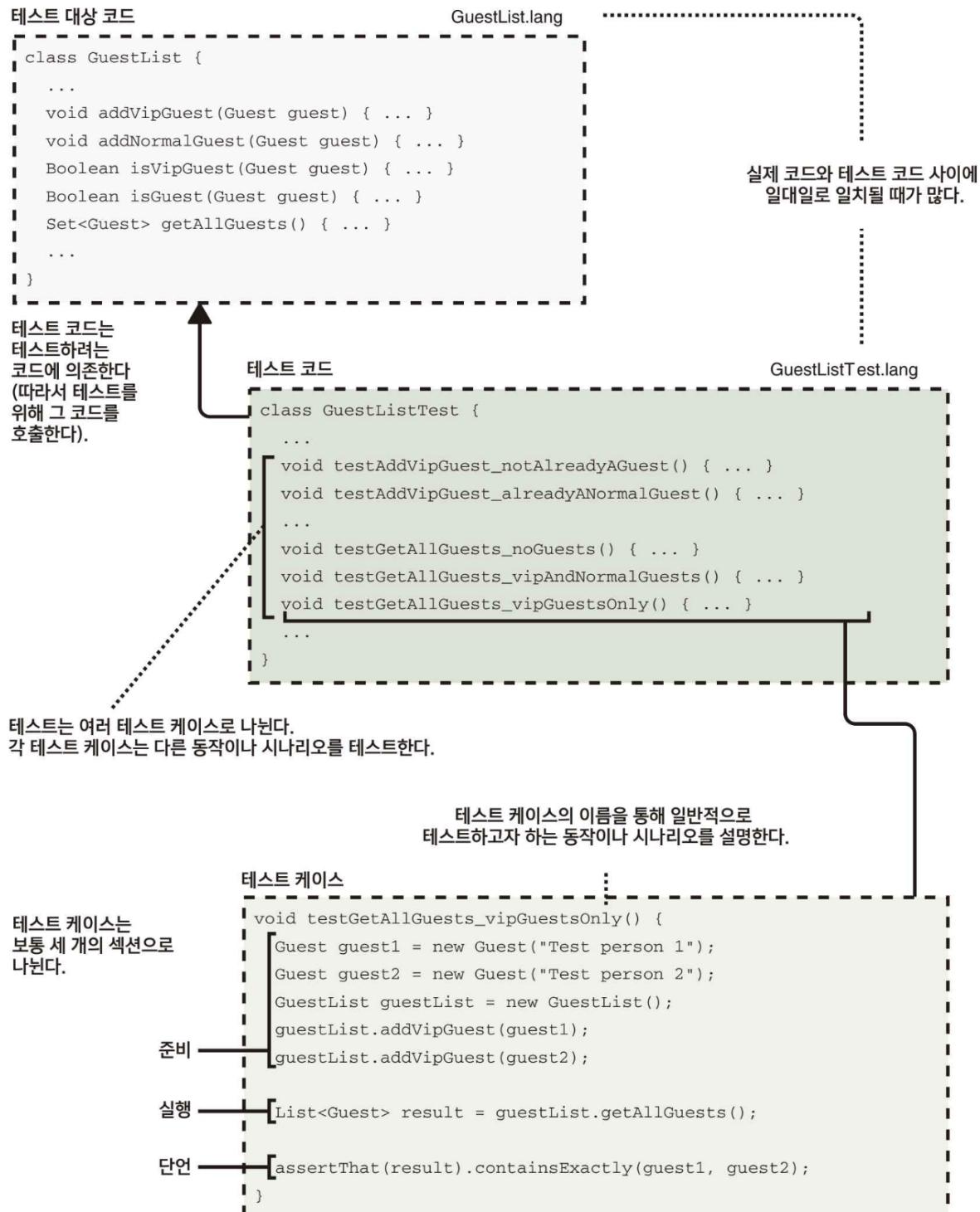


그림 10.1 다양한 단위 테스트 개념의 연관성

- **테스트 중인 코드 (code under test):** ‘실제 코드’라고도 하며, 테스트의 대상이 되는 코드입니다.
- **테스트 코드 (test code):** 단위 테스트를 구성하는 코드로, 일반적으로 실제 코드와는 별도의 파일에 존재합니다. 보통 일대일 매핑을 이루며, 예를 들어 `GuestList.lang`이라는 파일에 실제 코드가 있다면 테스트 코드는 `GuestListTest.lang`이라는 파일에 위치하게 됩니다.
- **테스트 케이스 (test case):** 각 파일에는 여러 테스트 케이스가 포함되어 있으며, 각 케이스는 특정 동작이나 시나리오를 테스트합니다. 테스트 케이스는 보통 세 가지 섹션으로 나뉩니다.

- **준비 (arrange)**: 테스트할 동작을 호출하기 전에 필요한 설정을 수행하는 단계입니다. 테스트 값 설정, 의존성 설정, 또는 클래스 인스턴스를 생성하는 작업이 포함됩니다.
- **실행 (act)**: 실제로 테스트 대상 코드를 호출하는 단계입니다.
- **단언 (assert)**: 실행 후 결과가 예상대로 나왔는지 확인합니다. 반환값이 예상값과 일치하거나, 특정 상태가 예상과 같은지 검증합니다.
- **given, when, then**

NOTE 주어진(given), 때(when), 그리고 나면(then)

일부 개발자는 준비, 실행, 단언이라는 용어보다 주어진(given), 때(when), 그리고 나면(then)이라는 용어를 선호한다. 테스트 철학에 따라 이러한 다른 용어를 사용하는 것을 옹호하는 듯한 뉘앙스가 있지만, 테스트 케이스 내의 코드의 맥락에서 보면 용어가 의미하는 바는 동일하다.

- **테스트 러너 (test runner)**: 테스트를 실제로 실행하는 도구입니다. 테스트 코드 파일을 받아 각 테스트 케이스를 실행하고, 통과하거나 실패한 케이스에 대한 결과를 출력합니다.

10.2 좋은 단위 테스트는 어떻게 작성할 수 있는가?

단위 테스트는 코드의 무결성을 검증하는 중요한 역할을 합니다. 그러나 잘못된 방식으로 작성하면, 유지 관리가 어렵고 예상치 못한 버그가 발생할 수 있습니다. 좋은 단위 테스트를 위해서는 다음과 같은 다섯 가지 주요 특성이 필요합니다.

10.2 절에서 말하는 내용은 다음 5가지가 다입니다.

1. 훼손을 정확하게 감지:

- 코드가 훼손되면 테스트가 실패해야 하고, 테스트는 코드가 실제로 훼손된 경우에만 실패해야 합니다.
- 코드가 변경되어도 의도된 대로 작동하는지 확인하는 것이 단위 테스트의 중요한 목적입니다. 테스트를 통해 신뢰성을 확보하고, 코드 병합 전에 오류를 발견하여 수정할 수 있습니다.
- 미래의 훼손을 방지하기 위해 회귀 테스트를 실행하여 정상적으로 동작하던 기능이 고장 나지 않도록 합니다.

2. 구현 세부사항에 독립적:

- 테스트 코드는 코드의 세부 구현이 변경되더라도 수정할 필요가 없어야 합니다.
- 세부사항이 변경될 때마다 테스트 코드를 수정해야 하는 것은 비효율적이므로, 높은 유지보수성을 위해 구현 세부사항에 의존하지 않도록 합니다.

3. 명확한 실패 설명:

- 테스트가 실패하면 그 원인을 명확히 알려야 합니다.
- 실패의 이유와 문제가 발생한 부분을 정확히 알 수 있어야 개발자가 신속히 오류를 수정할 수 있습니다.

4. 이해하기 쉬운 테스트 코드:

- 다른 개발자들도 쉽게 이해할 수 있는 코드로 작성해야 합니다.
- 테스트 코드의 의도와 수행 방식을 파악하기 쉬워야 합니다. 명확하게 작성된 테스트 코드는 코드베이스의 신뢰도를 높이고 유지보수를 용이하게 만듭니다.

5. 쉬운 실행과 빠른 속도:

- 테스트는 쉽게 실행할 수 있어야 하며, 빠르게 완료되어야 합니다.
- 느리거나 실행하기 어려운 테스트는 개발자가 자주 실행하지 않게 되므로, 일상적인 개발 작업에서 단위 테스트가 자주 실행될 수 있도록 해야 합니다.

10.2.1 훼손을 정확하게 감지하는 테스트

단위 테스트의 가장 기본적인 목적은 코드가 훼손되지 않았음을 확인하는 것입니다. 코드가 손상되었을 때, 테스트는 반드시 실패해야 하고, 코드는 철저한 테스트를 통해 병합 전에 오류를 잡아낼 수 있어야 합니다.

이러한 훼손 방지 기능은 다음 두 가지 중요한 역할을 합니다.

- **초기 신뢰 부여:** 코드 작성과 동시에 철저한 테스트를 통해 초기의 신뢰성을 보장하고, 병합 전 실수를 방지할 수 있습니다.
- **미래의 훼손 방지:** 코드베이스는 지속적으로 많은 개발자에 의해 변경되므로, 다른 개발자가 실수로 코드를 훼손할 가능성이 큽니다. 훼손이 발생할 때 테스트가 실패하도록 만들어야만 코드의 신뢰성을 유지 할 수 있습니다.
 - 코드 변경으로 인해 잘 돌아가던 코드가 작동하지 않는 것을 회귀라고 합니다.
 - 이러한 회귀를 탐지할 목적으로 테스트를 실행하는 것을 회귀 테스트라고 합니다.

이때 코드가 훼손된 경우에만 테스트가 실패해야 하는데, 이를 위반하는 경우 문제가 됩니다. 예를 들어, 테스트가 특정 조건에서 통과되었다가 다른 조건에서는 실패하는 **플래키(flakey)** 테스트가 여기에 해당합니다. 이는 무작위성, 타이밍 문제, 외부 시스템 의존성 등으로 인해 발생할 수 있습니다. 플래키 테스트는 개발자가 원인을 파악하는 데 시간을 낭비하게 할 뿐 아니라, 계속된 오류로 인해 테스트의 신뢰를 잃게 합니다.

즉, 코드 훼손 시에만 테스트가 실패하도록 작성하는 것은 단위 테스트의 신뢰성을 유지하기 위해 매우 중요합니다.

10.2.2 세부 구현 사항에 독립적

개발자가 코드에 가할 수 있는 변경은 크게 두 가지로 나뉩니다.

1. **기능적 변화:** 이는 코드의 외부 동작을 변경하는 것으로, 새로운 기능 추가, 버그 수정, 에러 처리 등이 포함됩니다. 기능적 변경은 코드를 사용하는 모든 사용자에게 영향을 미칠 수 있으므로, 코드 변경 전 후 출하는 쪽에서 신중히 고려해야 합니다. 이 변경은 코드의 동작을 수정하기 때문에, 테스트도 수정을 요구할 것으로 예상됩니다. 만약 테스트 수정이 필요 없다면, 원래 테스트가 충분하지 않았을 가능성이 있습니다.

2. 리팩터링: 리팩터링은 코드의 구조를 변경하지만, 외부에서 보이는 동작에는 영향을 미치지 않습니다. 즉, 코드의 구현 세부 사항을 바꾸되 사용자에게 주의가 필요한 변화는 없습니다. 하지만 코드 수정을 통해 실수로 외부 동작이 바뀌지 않았다는 것을 어떻게 확인할 수 있을까요?

리팩터링을 통해 코드의 외부 동작이 의도치 않게 변경되지 않았음을 확신하기 위해, 테스트 코드를 작성할 때 두 가지 접근 방식을 고려해볼 수 있습니다.

접근 방식

- 접근 방식 A: 코드의 동작뿐만 아니라 다양한 구현 세부 사항까지 확인하는 방법입니다. 프라이빗 함수나 멤버 변수를 직접 조작해 내부 상태를 검증합니다.

```
import static org.junit.jupiter.api.Assertions.*;  
  
import org.junit.jupiter.api.Test;  
import java.lang.reflect.Field;  
  
public class BankAccountTest {  
  
    @Test  
    public void testDeposit() throws NoSuchFieldException,  
IllegalAccessException {  
        BankAccount account = new BankAccount(100);  
        account.deposit(50);  
  
        // Reflection을 사용하여 private balance 필드에 접근  
        Field balanceField = BankAccount.class.getDeclaredField("balance");  
        balanceField.setAccessible(true);  
        int balance = (int) balanceField.get(account);  
  
        assertEquals(150, balance); // 내부 balance 값을 직접 검증  
    }  
  
    @Test  
    public void testWithdraw() throws NoSuchFieldException,  
IllegalAccessException {  
        BankAccount account = new BankAccount(100);  
        account.withdraw(50);  
  
        // Reflection을 사용하여 private balance 필드에 접근  
        Field balanceField = BankAccount.class.getDeclaredField("balance");  
        balanceField.setAccessible(true);  
        int balance = (int) balanceField.get(account);  
  
        assertEquals(50, balance); // 내부 balance 값을 직접 검증  
    }  
}
```

```
}
```

```
}
```

- 접근 방식 **B**: 동작만을 테스트하고, 구현 세부 사항에는 의존하지 않습니다. 오직 공개된 API를 통해 상태를 설정하고 결과를 확인하며, 프라이빗 함수나 변수에는 접근하지 않습니다.

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class BankAccountTest {

    @Test
    public void testDeposit() {
        BankAccount account = new BankAccount(100);
        account.deposit(50);
        assertEquals(150, account.getBalance()); // 외부 동작만 확인
    }

    @Test
    public void testWithdraw() {
        BankAccount account = new BankAccount(100);
        account.withdraw(50);
        assertEquals(50, account.getBalance()); // 외부 동작만 확인
    }
}
```

리팩터링 시 두 접근 방식의 결과

- 접근 방식 **A**: 리팩터링을 올바르게 수행했어도, 프라이빗 함수나 내부 변수의 변화로 인해 테스트가 실패할 가능성이 큽니다. 따라서 테스트 코드의 많은 부분을 수정해야 하며, 어디서 실패가 발생했는지 파악하기 어렵습니다.
- 접근 방식 **B**: 리팩터링이 제대로 수행됐다면 테스트 코드 수정 없이 통과할 것입니다. 만약 테스트가 실패하면, 이는 리팩터링 중 실수로 외부 동작이 변경되었음을 의미합니다.

기능 변경과 리팩터링을 같이 하지 말라

코드베이스를 변경할 때 일반적으로 기능만 변경하거나 리팩터링만 해야지 두 가지 작업을 동시에 수행하는 것은 좋지 않다.

리팩터링은 어떠한 동작도 변경하지 않지만, 기능 변경은 동작을 변경한다. 기능적 변화와 리팩터링을 동시에 하면 기능적 변화로 예상되는 동작의 변화와 리팩터링의 실수로 발생하는 동작의 변화를 구분하기 어려울 수 있다. 보통 리팩터링을 한 다음 기능 변경을 따로 하는 것이 좋다. 이렇게 하면 잠재적인 문제의 원인을 분리하기가 훨씬 더 쉬워진다.

따라서 테스트는 외부 동작을 검증하는 데 집중하고, 구현 세부 사항에 의존하지 않도록 작성하는 것이 좋습니다. 이를 통해 리팩터링 및 유지보수 시에도 신뢰할 수 있는 테스트 결과를 얻을 수 있고, 테스트 코드의 유지보수 부담도 줄어듭니다.

10.2.3 잘 설명되는 실패

테스트의 주요 목적 중 하나는 미래의 변경으로부터 코드의 안정성을 보호하는 것입니다. 일반적으로, 다른 개발자가 작성한 코드에 대해 자신이 수정한 코드가 의도하지 않게 영향을 줄 수 있으며, 이때 테스트는 실패를 통해 문제가 발생했음을 개발자에게 알려줍니다. 개발자는 테스트 결과를 확인하고 무엇이 문제인지 빠르게 파악할 수 있습니다. 하지만 테스트 실패가 구체적이지 않으면, 문제 원인을 알아내는 데 시간이 낭비될 수 있습니다.

실패에 대한 자세한 내용을
보여주지 않는 테스트 실패

테스트 케이스의 이름으로부터
어떤 동작을 테스트하는지 알 수 없다.

```
Test case testGetEvents failed:  
Expected: [Event@ea4a92b, Event@3c5a99da]  
But was actually: [Event@3c5a99da, Event@ea4a92b]
```

실패 메시지가 이해하기 어렵다.

실패에 대한 자세한 내용을
잘 설명해주는 테스트 실패

테스트 케이스의 이름으로부터
어떤 동작이 테스트되고 있는지 알 수 있다.

```
Test case testGetEvents_inChronologicalOrder failed:  
Contents match, but order differs  
Expected:  
[<Spaceflight, April 12, 1961>, <Moon Landing, July 20, 1969>]  
But was actually:  
[<Moon Landing, July 20, 1969>, <Spaceflight, April 12, 1961>]
```

실패 메시지가 명확하다.

그림 10.2 단지 잘못됐다고만 알려주는 테스트보다 무엇이 잘못되었는지 명확하게 설명하는 테스트 실패가 훨씬 더 유용하다.

좋은 테스트 실패 메시지의 중요성

- 테스트가 문제 발생 시 구체적이고 명확하게 실패 원인을 알려주는 메시지를 제공하는 것이 중요합니다. 명확한 메시지가 없다면 개발자는 무엇이 잘못되었는지 알아내기 위해 많은 시간을 소비하게 됩니다.
- 예를 들어, 첫 번째 메시지가 “이벤트 수신이 실패했다”고만 표시하는 대신, 두 번째 메시지는 “받은 이벤트가 시간 순서대로 되지 않음”이라는 구체적인 설명을 제공하여 문제 원인을 더 쉽게 파악하게 합니다.

효과적인 테스트 실패 메시지를 위한 방법

1. 하나의 테스트 케이스에서는 한 가지 항목만 검사하도록 작성하는 것이 좋습니다. 이를 통해, 무엇이 잘 못되었는지를 더욱 명확히 알 수 있습니다.
2. 테스트 케이스에 서술적인 이름을 사용하면, 실패한 테스트 이름만으로도 문제가 되는 동작을 빠르게 이해할 수 있습니다.

이렇게 하면, 여러 가지를 한꺼번에 테스트하는 큰 케이스보다 각각의 작은 테스트 케이스가 개별 동작을 확인하도록 작성되기 때문에, 문제가 발생한 부분을 정확히 파악할 수 있다고 합니다.

10.2.4 이해 가능한 테스트 코드

일반적으로 테스트가 실패하면 코드가 제대로 작동하지 않는다고 생각할 수 있습니다. 하지만 정확히 말하면, 테스트 실패는 코드가 예상과 다른 방식으로 작동하고 있음을 나타내는 것일 뿐입니다. 코드의 변경이 실제로 문제를 일으킨 것인지 아니면 의도된 동작인지 상황에 따라 다를 수 있습니다. 예를 들어, 개발자가 새로운 요구 사항에 맞춰 코드를 의도적으로 수정했다면 동작의 변화는 의도된 것입니다.

개발자는 변경된 코드가 안전한지 확인한 뒤, 그 변화가 새로운 기능 요구 사항을 반영하는 것이라면 테스트 코드 또한 수정해야 합니다. 예를 들어, 코드에서 세 가지 기능을 테스트하고 있는데, 그 중 하나만 의도적으로 변경되었다면 해당 기능에 대한 테스트 케이스만 업데이트하고 나머지 두 가지 기능에 대한 테스트는 그대로 두는 것이 이상적입니다. 이를 위해서는 각각의 테스트 케이스가 무엇을 테스트하는지 명확하게 이해하고 있어야 합니다.

테스트 코드 이해의 중요성

테스트 코드가 이해하기 어려운 경우 두 가지 일반적인 문제 상황이 발생할 수 있습니다:

1. 너무 많은 것을 한 번에 테스트하는 경우: 많은 기능을 한 번에 테스트하면 테스트의 목적이 불분명해지기 때문에, 어떤 코드 변경이 안전한지 이해하기 어렵습니다.
2. 과도한 테스트 설정 공유: 공통 설정이 과하게 공유되면 테스트 로직이 복잡해져서 특정 기능이 올바르게 동작하는지 확인하는 데 어려움을 겪을 수 있습니다.

따라서 테스트 코드는 이해하기 쉽게 작성해야 하며, 각각의 테스트가 무엇을 테스트하는지 명확하게 설명되어야 한다고 합니다.

10.2.5 쉽고 빠른 실행

대부분의 단위 테스트는 자주 실행됩니다. 단위 테스트의 주요 목적 중 하나는 잘못된 코드가 코드베이스에 병합되는 것을 방지하는 것입니다. 이를 위해 많은 코드베이스에서는 병합 전 테스트를 통과해야만 병합이 가능하도록 설정합니다. 하지만 테스트 실행에 오랜 시간이 걸리면 코드 변경이 작거나 사소한 경우에도 모든 개발자의 작업 속도가 느려질 수 있습니다.

단위 테스트 속도의 중요성

단위 테스트는 코드베이스 병합 전뿐만 아니라 개발 중에도 자주 실행되므로, 느린 단위 테스트는 개발자의 작업 속도를 저하시킵니다. 단위 테스트는 빠르고 간단하게 유지해야 하는 또 다른 이유는 개발자가 테스트를 자

주 실행할 수 있게 해주기 위해서입니다. 테스트가 느리고 실행하기 어려우면 테스트를 하지 않으려는 경향이 생길 수 있기 때문입니다.

빠르고 쉬운 테스트가 가져오는 효과

테스트가 쉽고 빠르게 실행 가능할수록 개발자는 더 효율적으로 작업할 수 있으며, 테스트의 빈도와 범위도 넓어지므로 더 철저하게 코드를 검증할 수 있습니다.

이 부분은 CI/CD 시 테스트를 돌리는 걸 생각하시면 될 것 같습니다. 테스트가 너무 오래걸리면 병합하는데 속도가 너무 걸려서 작업에 지연이 생기겠죠?

느낀점

테스트는 외부 동작을 검증하는 데 집중하고, 구현 세부 사항에 의존하지 않도록 작성하라, 하나의 테스트 케이스에서는 한 가지 항목만 검사하라 이 2가지가 기억에 남습니다.

나머지는 어떻게 보면 당연하다고 여겨질 수도 있는 부분이라 위 2가지를 생각하며 테스트 코드를 작성하면 좋을 것 같아요!

10.3 퍼블릭 API에 집중하되 중요한 동작은 무시하지 말라

단위 테스트의 주요 목표 중 하나는 구현 세부 사항에 의존하지 않고, 외부 API를 통한 동작을 검증하는 것입니다. 이는 코드가 공개 API와 구현 세부 사항으로 나눌 수 있다는 개념에 기초합니다.

테스트 시 공개 API에만 집중하는 것이 좋다는 일반적인 권장사항입니다. 공개 API에 초점을 맞추면, 코드 사용자 입장에서 중요한 동작을 테스트하는 데 집중할 수 있고, 구현 세부 사항은 코드 목적을 달성하는 수단일 뿐이기 때문입니다.

```
Double calculateKineticEnergyJ(Double massKg, Double speedMs) {  
    return 0.5 * massKg * Math.pow(speedMs, 2.0);  
}
```

위 함수는 운동 에너지를 줄(joule) 단위로 계산하며, 사용자는 이 함수가 질량(mass)과 속도(speed)를 입력 받아 정확한 값을 반환할 것임을 기대합니다. 여기서 Math.pow() 함수의 호출 여부는 구현 세부 사항에 해당하며, 이를 사용하든 speedMs * speedMs를 사용하든 결과는 동일하므로 실제 함수 동작에는 영향을 미치지 않습니다.

```
void testCalculateKineticEnergy_correctValueReturned() {  
    assertThat(calculateKineticEnergyJ(3.0, 7.0))  
        .isWithin(1.0e-10)  
        .of(73.5);  
}
```

값이 73.5의 0.0000000001 이내인지 확인한다.

구현 세부 사항과 무관하게 테스트하기

테스트에서 공개 API에 집중하면, 입력값과 기대되는 출력값을 중심으로 테스트 케이스를 작성할 수 있습니다. 예를 들어, calculateKineticEnergyJ()가 주어진 질량과 속도를 입력받았을 때 예상한 운동 에너지 값을 정확히 반환하는지 확인하는 테스트가 가능합니다. Math.pow() 함수의 호출 여부는 중요하지 않으므로 테스트에 포함할 필요가 없습니다.

이렇게 함으로써 구현 세부 사항에 의존하지 않으면서도 핵심 동작을 검증할 수 있으며, 코드의 변경이 필요할 때 구현만 바꾸고 테스트는 그대로 유지할 수 있습니다.

10.3.1 중요한 동작이 퍼블릭 API 외부에 있을 수 있다.



그림 10.3 커피 자판기에는 공용 API가 있지만, 공용 API만으로는 완벽하게 테스트할 수 없다.

현실에서의 코드는 종종 독립적이지 않으며, 다양한 외부 요소에 의존하는 경우가 많습니다. 이때 “**공용 API만을 이용한 테스트**” 원칙이 적용되지 않는 경우가 생깁니다. 예를 들어, 다른 코드에 의존하거나 외부 입력이 필요한 경우, 또는 의존하는 코드에 부수적인 효과가 발생하면 테스트 방식에 영향을 미칩니다. 이런 상황에서 공용 API를 넘어서는 테스트가 필요할 수 있습니다.

일반적으로 **공용 API**는 코드 사용자가 접근하는 주요 기능을 의미합니다. 이 원칙은 공용 API에 집중해 테스트를 작성하는 것이지만, 현실에서는 공용 API로만 테스트하기 어려운 경우가 존재합니다. 예를 들어, 커피 자판기 테스트에서는 기계가 고객에게 보여주는 주요 동작 외에도 설정 및 작동 조건에 대한 테스트가 필요합니다.

예시: 커피 자판기 테스트

1. 고객이 상호작용하는 공용 API: 신용카드 인식, 커피 선택, 결제 후 커피 제공과 같은 고객이 직접 경험하는 기능을 의미합니다.
2. 필요한 설정 및 의존성: 커피 자판기의 테스트를 위해서는 전원, 물, 커피콩 등 준비가 필요합니다. 고객 입장에서는 이는 구현 세부 사항이지만, 테스트에서는 필수적인 준비 작업입니다.
3. 의도된 부수 효과: 예를 들어, 스마트 자판기는 물이나 커피콩이 떨어지면 담당자에게 알림을 보내는 부수적인 효과가 있습니다. 이는 고객에게는 구현 세부 사항이지만, 테스트에서는 중요한 동작입니다.

구현 세부 사항의 중요성 여부 판단

모든 구현 세부 사항이 테스트 대상이 되는 것은 아닙니다. 예를 들어, 물이 끓는 방식(열전 차단기 vs. 보일러)은 커피의 맛에 영향을 미칠 수 있지만, 이는 테스트 대상이 아닌 구현 세부 사항으로 간주됩니다. 궁극적으로 테스트해야 할 것은 커피의 맛이며, 물 끓이는 방식은 그저 커피 맛을 위한 수단일 뿐입니다.

예제 10.1 AddressBook 클래스

```
class AddressBook {  
    private final ServerEndPoint server;  
    private final Map<Int, String> emailAddressCache;  
    ...
```

클래스 사용자에 관련된
구현 세부 사항

```
String? lookupEmailAddress(Int userId) {  
    String? cachedEmail = emailAddressCache.get(userId);  
    if (cachedEmail != null) {  
        return cachedEmail;  
    }  
    return fetchAndCacheEmailAddress(userId);  
}
```

퍼블릭 API

```
private String? fetchAndCacheEmailAddress(Int userId) {  
    String? fetchedEmail = server.fetchEmailAddress(userId);  
    if (fetchedEmail != null) {  
        emailAddressCache.put(userId, fetchedEmail);  
    }  
    return fetchedEmail;  
}
```

보다 자세한
구현 세부 사항

테스트는 가능하다면 퍼블릭 API를 사용해서만 테스트하는 것을 목표로 해야 한다.

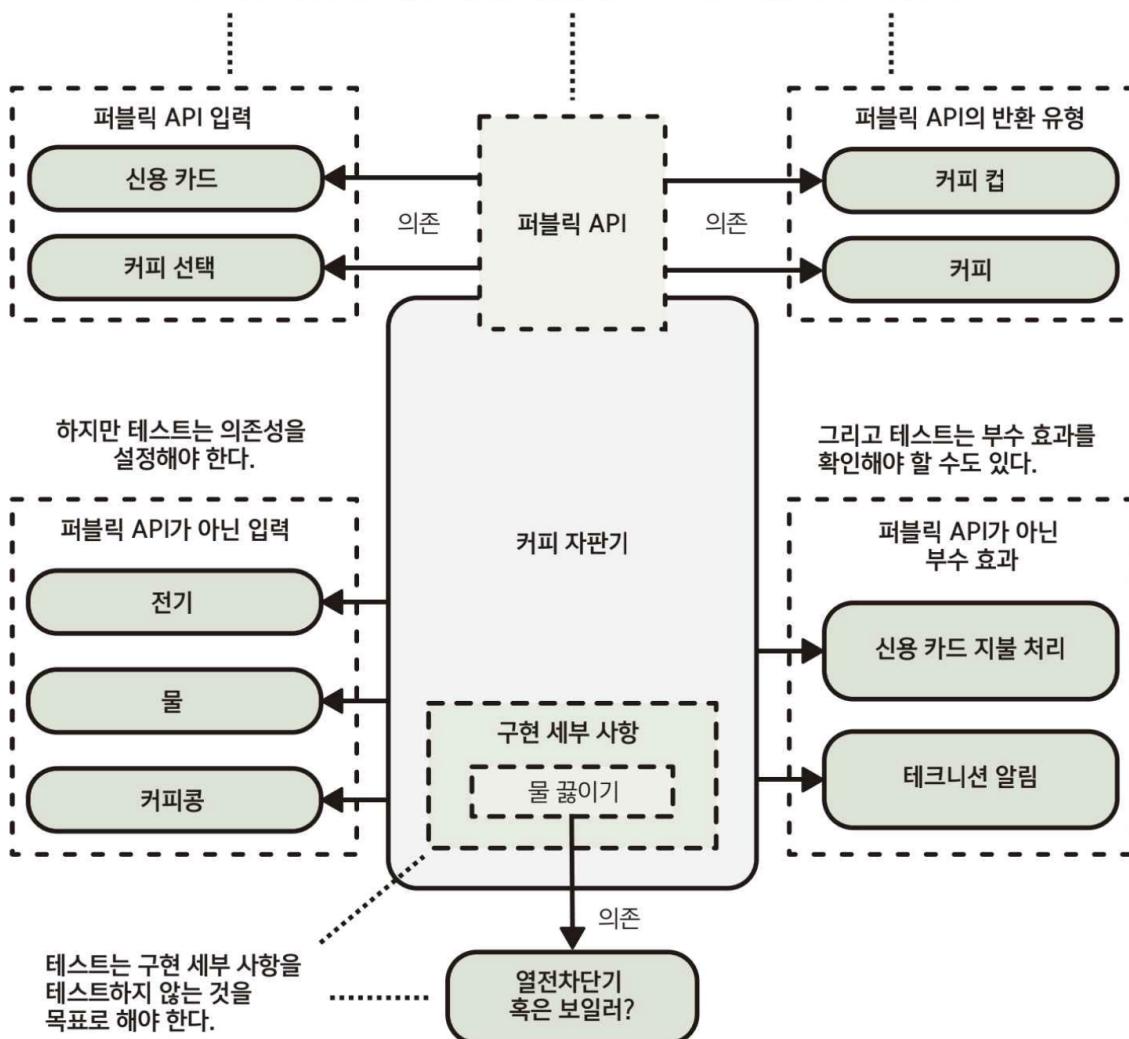


그림 10.4 테스트는 가능하면 퍼블릭 API를 사용하여 테스트하는 것을 목표로 해야 한다. 그러나 설정을 수행하고 원하는 부수 효과를 확인하기 위해 테스트가 공용 API의 일부가 아닌 종속성과 상호작용해야 하는 경우가 많다.

일반적으로 **Public API**를 통해 코드를 테스트하는 것이 가장 바람직합니다. Public API는 사용자나 외부 시스템이 접근할 수 있는 주요 기능을 제공하며, 이를 통해 코드의 주요 동작을 검증하면 실제로 코드가 의도대로 작동하는지 확인할 수 있습니다. Public API를 이용한 테스트는 사용자가 신경 써야 할 핵심 동작에만 초점을 맞추고, 내부 구현 세부 사항에는 의존하지 않으므로 테스트 코드 유지보수가 쉬워집니다.

그러나 Public API만으로 모든 동작을 검증하기 어려운 경우도 있습니다. 예를 들어, 코드가 서버와 상호작용하거나 데이터베이스와 연결되는 경우, 테스트 대상 코드가 내부적으로 의존하는 부분을 설정하거나 시뮬레이션해야 하는 상황이 생길 수 있습니다. 예를 들어, AddressBook 클래스가 사용자의 이메일 주소를 조회할 때 서버와 통신하는 경우, 캐싱 기능이 포함된 이 코드에서 동일한 요청을 반복할 때 서버 호출이 발생하지 않도록 테스트할 필요가 있습니다. 하지만 캐싱 기능은 Public API에 드러나지 않기 때문에 이를 검증하려면 캐시 사용 여부를 구현 세부 사항으로 검증해야 합니다.

또한, 중요한 부수 효과가 발생하는지 확인하는 경우에도 Public API만으로는 한계가 있을 수 있습니다. 이럴 때는 필요한 부수 효과를 검증하기 위해 특정 구현 세부 사항을 테스트해야 할 수도 있습니다

결론적으로, 가능하면 Public API를 통해 주요 동작을 검증하는 것이 가장 좋지만, 특정 상황에서는 구현 세부 사항을 테스트할 수밖에 없는 경우가 있습니다. 중요한 것은 테스트 코드가 가능한 한 구현 세부 사항과 독립적이도록 유지하되, 불가피한 경우에만 Public API 범위를 벗어나서 테스트해야 한다는 점입니다.

추가(private 메서드 테스트 하는 방법!)

private 메서드를 직접 테스트하려는 상황이 생길 때, 이를 우회하고도 주요 동작을 검증하는 몇 가지 방법이 있다고 합니다.

1. Public API를 통한 간접 테스트:

- 가능한 한 public 메서드를 통해 private 메서드의 동작을 검증할 수 있다고 합니다. private 메서드는 보통 내부 로직을 위한 것이기 때문에, public 메서드의 호출을 통해서도 그 내부 private 메서드의 동작 결과를 확인할 수 있어야 합니다. 이를 통해 private 메서드의 세부 사항에 의존하지 않으면서도 올바르게 테스트할 수 있습니다.

2. 리팩토링을 통해 private 메서드의 역할 재배치:

- private 메서드가 복잡한 로직을 포함하고 여러 곳에서 사용된다면, 별도의 클래스로 분리하여 public 메서드로 만들 수도 있습니다. 이를 통해 테스트 가능성은 높이고, 코드 재사용성을 향상시키며, 객체의 책임을 분리할 수 있습니다.

3. 패키지-프라이빗(package-private) 접근 제어자 사용:

- 꼭 필요한 경우, 테스트를 위해 private 메서드를 패키지-프라이빗(접근 제어자를 사용하지 않음)으로 선언하여 테스트 코드에서 접근할 수 있게 할 수도 있습니다. 이는 테스트 클래스가 같은 패키지에 있을 때만 접근을 허용합니다. 다만, 이 방법은 매우 제한적으로 사용하며, 핵심 기능이 아닌 보조적 기능일 때 적합합니다.

4. Reflection 사용:

- 최후의 수단으로 reflection을 사용하여 private 메서드를 호출할 수 있지만, 이는 일반적으로 권장되지 않습니다. Reflection을 사용하면 코드의 가독성과 유지보수가 떨어지기 때문에, 부득이한 경우에만 사용하는 것이 좋습니다.

여기서 가장 적합한 방식은 가장 적합한 방식은, 구현 세부 사항에 대한 의존성을 피할 수 있는 **Public API**를 통한 간접 테스트 라고 합니다!!

10.4 테스트 더블

단위 테스트에서 코드를 격리하여 테스트하려면 의존성을 직접 사용할 수 없는 경우가 있습니다. 테스트 대안으로 ‘테스트 더블 (test double)’이 유용한데, 테스트 더블은 의존성을 시뮬레이션하여 테스트에 더 적합하게 만들어진 객체입니다.

테스트 더블을 사용하는 이유와 유형을 간략히 정리하면 다음과 같습니다.

1. 테스트 더블의 필요성:

- 코드를 격리하여 테스트하기 위해 실제 의존성을 사용하는 것이 어렵거나 바람직하지 않은 경우가 많습니다.
- 외부 시스템에 대한 종속성을 제거하고, 테스트 중 의존성에서 발생할 수 있는 부작용을 방지하는 데 유용합니다.

2. 테스트 더블의 세 가지 유형:

- 목(Mock)**: 호출 횟수, 호출된 메서드 등의 정보를 확인하여 특정 동작을 검증하는 데 사용합니다.
- 스텁(Stub)**: 미리 정해진 값을 반환하여 동작을 단순히 확인할 때 유용합니다.
- 페이크(Fake)**: 실제 동작과 비슷하게 구현되어 있으나, 테스트 목적에 맞게 단순화된 객체입니다.

이 10.4 장에서는 테스트 더블을 사용하는 이유를 설명하고, 테스트 더블 중 목과 스탬프이 문제가 될 수 있는 부분에 대해 설명하고, 왜 가능하면 페이크를 사용하는 것이 나은지를 설명해줍니다.

10.4.1 테스트 더블을 사용하는 이유

테스트 더블을 사용하는 일반적인 이유는 다음 세 가지입니다.

1. 테스트 단순화:

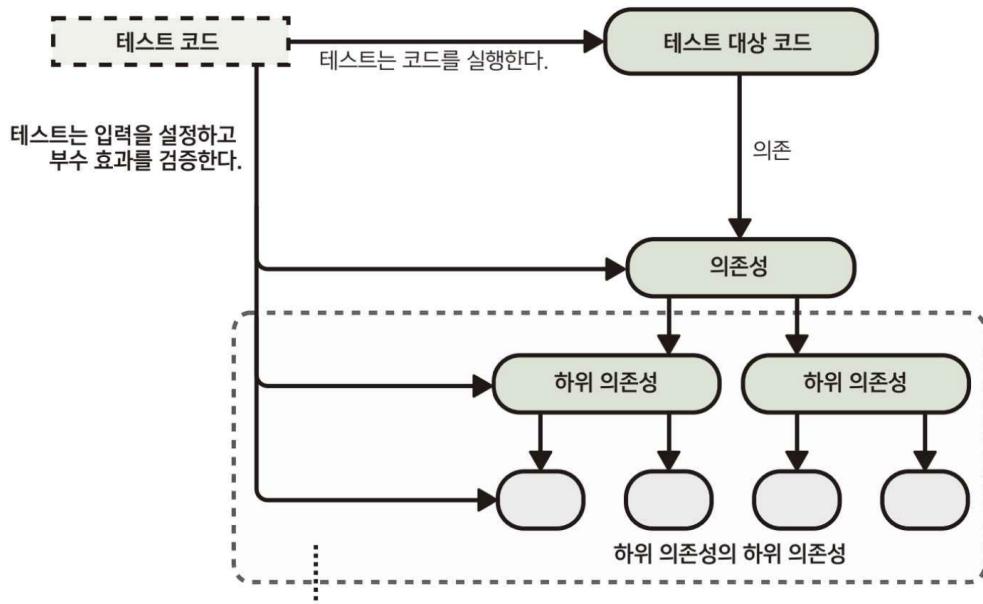


그림 10.6 테스트에서 의존성을 실제로 사용하는 것은 비현실적이다. 의존성이 하위 의존성을 많이 가지고 있고 이 하위 의존성과 상호작용이 필요한 경우가 이에 해당한다.

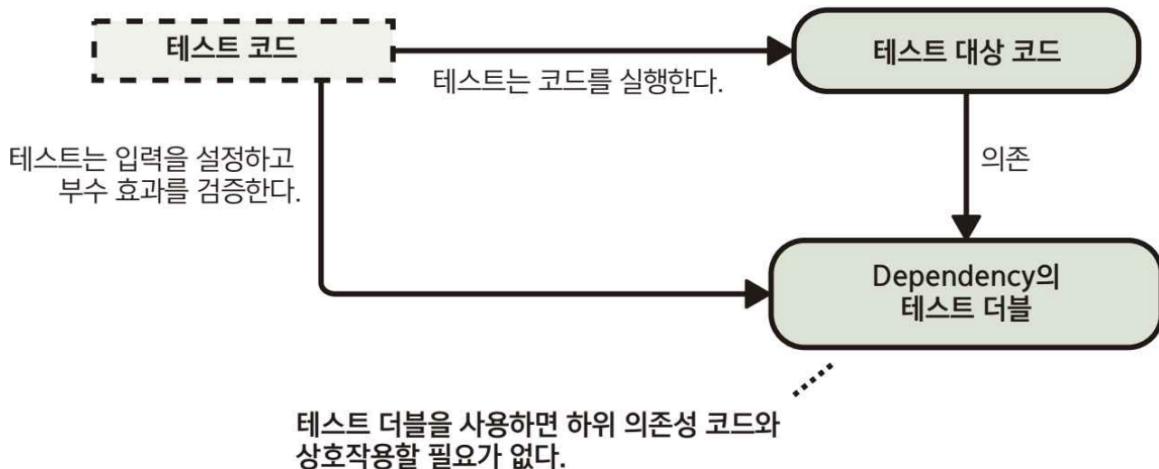


그림 10.7 테스트 더블은 하위 의존성에 대한 우려를 제거하여 테스트를 단순화할 수 있다.

- 일부 의존성은 설정이 복잡하여 테스트하기 어렵고 시간이 많이 듭니다. 예를 들어, 의존성의 하위 설정 까지 필요하거나 테스트와 구현 세부 사항이 밀접하게 결합될 수 있습니다. 이때, 실제 의존성 대신 테스트 더블을 사용하면 테스트가 단순해지고 테스트 코드를 더 효율적으로 작성할 수 있습니다.

2. 테스트로부터 외부 시스템 보호:

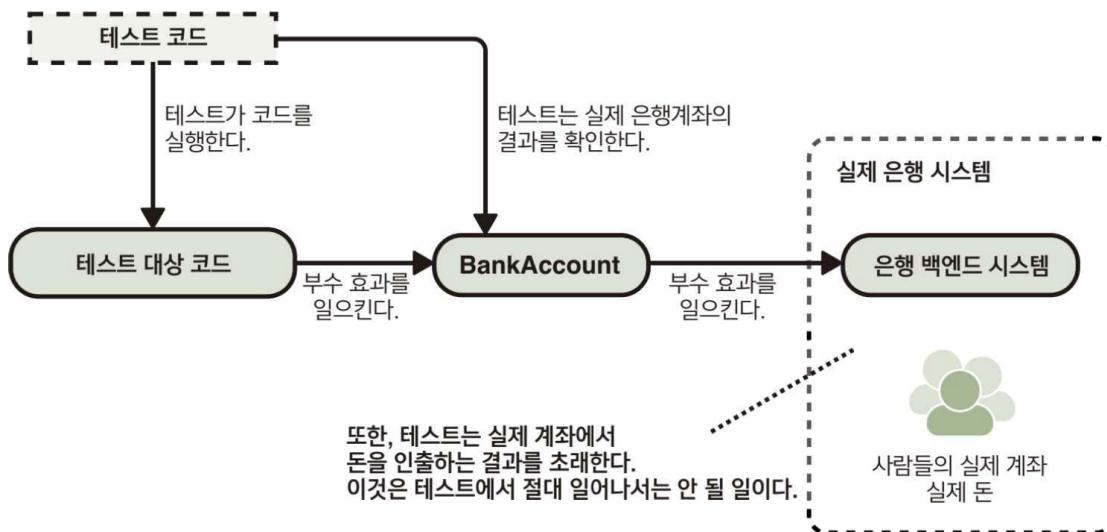


그림 10.8 의존성이 부수 효과를 실제로 일으킨다면, 의존성을 실제로 사용하는 대신 테스트 더블을 사용하는 것이 좋다.

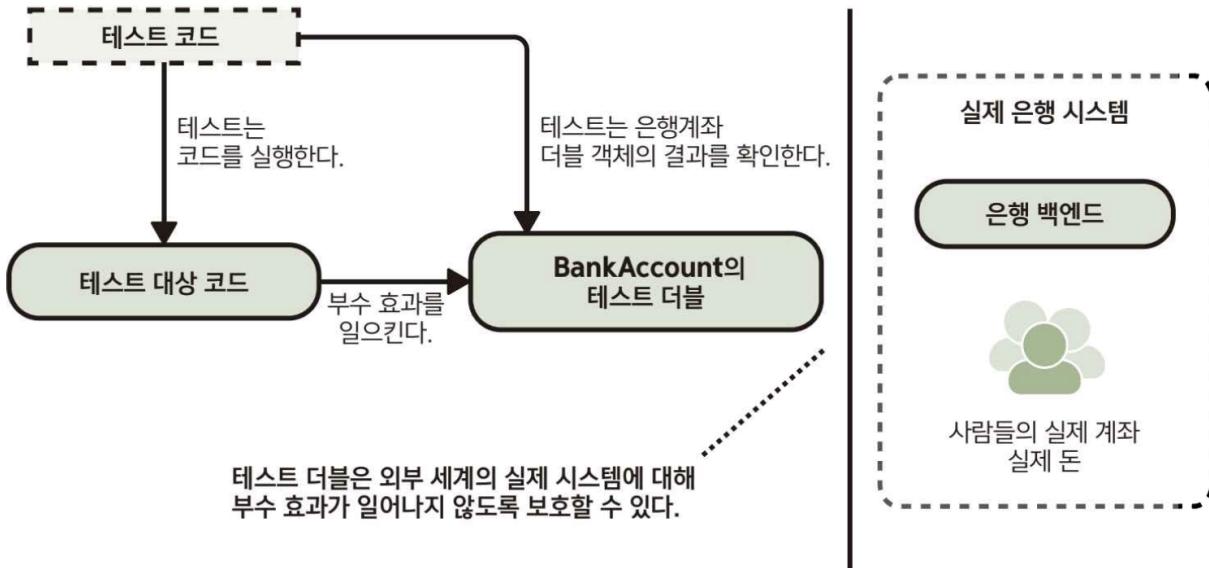


그림 10.9 테스트 더블은 외부 세계의 실제 시스템에 대해 부수 효과가 일어나지 않도록 보호할 수 있다.

- 일부 의존성은 실제 서버 요청이나 데이터베이스 수정과 같은 부수적인 결과를 초래할 수 있습니다. 이러한 경우 테스트 더블을 사용하면, 테스트가 외부 시스템에 영향을 주지 않고 독립적으로 실행될 수 있습니다. 이를 통해 시스템이 실수로 중요한 프로세스를 방해하지 않도록 보호할 수 있습니다.

3. 외부로 부터 테스트 보호:

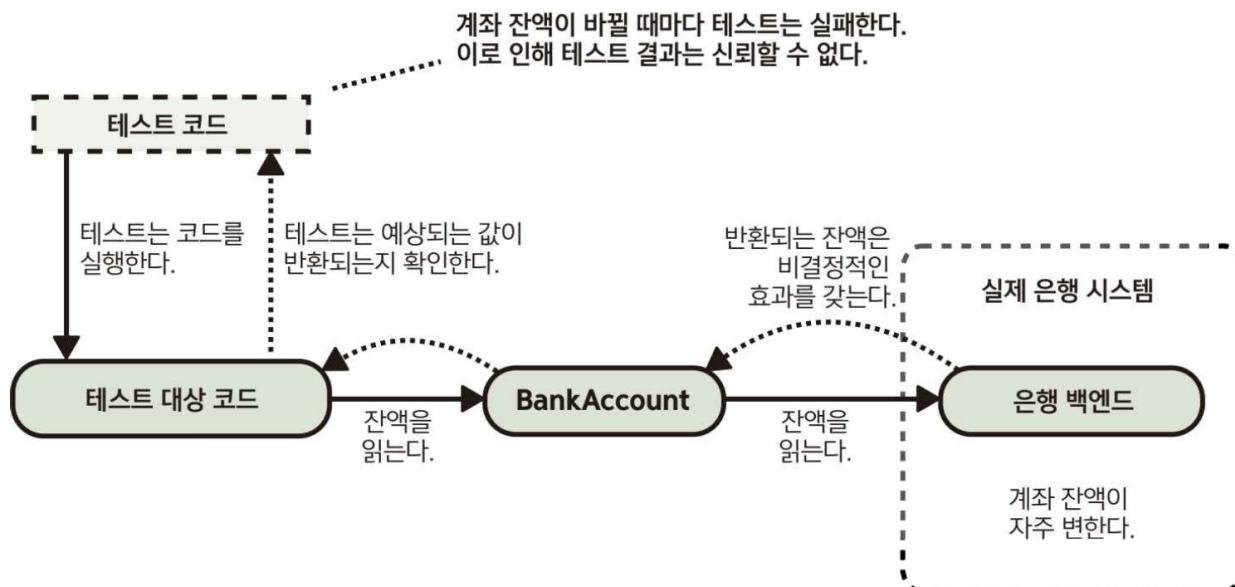


그림 10.10 의존성이 비결정적인 방식으로 작용하는 경우 테스트가 신뢰하기 어려워진다.

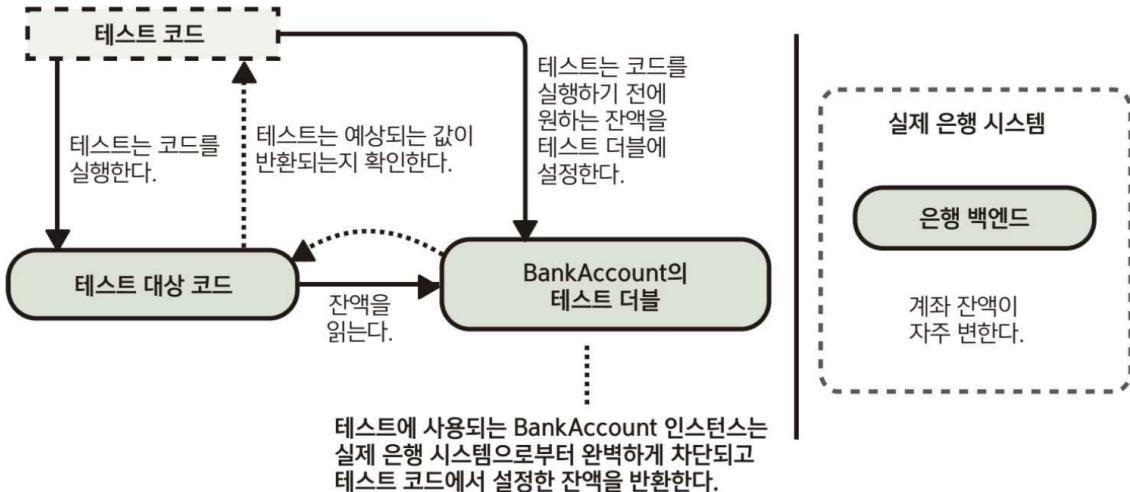


그림 10.11 테스트 더블은 의존성 코드가 실제로 동작할 때 일어날 수 있는 비결정적 동작으로부터 테스트를 보호한다.

- 외부 의존성은 비결정적일 수 있습니다. 예를 들어, 다른 시스템에 의해 데이터베이스 값이 수정되면 테스트 결과가 일정하지 않게 변할 수 있습니다. 이때 테스트 더블을 사용하면, 항상 동일한 결과가 나와 테스트의 신뢰성을 보장할 수 있습니다.

10.4.2 목

Mock 객체는 클래스나 인터페이스를 시뮬레이션하며, 함수 호출 시 전달된 매개변수를 기록하는 기능을 제공합니다. Mock은 테스트 대상 코드에서 필요한 함수가 의존성을 통해 올바르게 호출되는지를 검증하는 데 유용합니다.

예제 10.2 BankAccount에 의존하는 코드

```
class PaymentManager {
    ...
    PaymentResult settleInvoice(
        BankAccount customerBankAccount,
        Invoice invoice) {
        customerBankAccount.debit(invoice.getBalance());
        return PaymentResult.paid(invoice.getId());
    }
}
```

BankAccount 인스턴스를
매개변수로 받는다.

계좌로부터 청구서의 잔액만큼 인출하는 것은
테스트해야 할 동작 중 하나다.

은행 계좌 예제에서 PaymentManager 클래스는 settleInvoice() 메서드를 통해 고객의 은행 계좌에서 청구서 금액만큼을 인출해 결제합니다.

이 테스트에서 고객의 BankAccount는 의존성 객체로 사용되며, 정확한 금액이 인출되었는지 확인하는 것이 중요합니다. Mock 객체를 사용하여 BankAccount와의 상호작용을 기록하고, 테스트 대상 메서드가 실제 은행 계좌에 영향을 주지 않고도 부수 효과를 발생시키는지 확인할 수 있습니다.

예제 10.3 BankAccount 인터페이스 및 구현

```
interface BankAccount {  
    void debit(MonetaryAmount amount);  
    void credit(MonetaryAmount amount);  
    MonetaryAmount getBalance();  
}  
  
class BankAccountImpl implements BankAccount {  
    private final BankingBackend backend;  
    ...  
  
    override void debit(MonetaryAmount amount) { ... }  
    override void credit(MonetaryAmount amount) { ... }  
    override MonetaryAmount getBalance() { ... }  
}
```

BankingBackend에 의존하는데,
실제 은행계좌에 있는 잔액에 영향을 미친다.

BankAccount는 인터페이스이며, 이를 구현하는 클래스는 BankAccountImpl입니다. BankAccountImpl은 실제 은행 시스템과 연결되는 BankingBackend에 의존하므로, 테스트에서 이 인스턴스를 사용할 경우 실제 계좌에서 금액이 이동할 위험이 있습니다. 따라서, 외부 환경을 보호하기 위해 테스트에서 BankAccountImpl을 직접 사용해서는 안 됩니다.

대신, BankAccount 인터페이스의 Mock 객체를 생성하여 사용합니다. 이 Mock 객체를 통해 debit() 메서드가 예상한 금액으로 호출되었는지 검증합니다.

예제 10.4 목을 사용하는 테스트 케이스

```
void testSettleInvoice_accountDebited() {  
    BankAccount mockAccount = createMock(BankAccount);  
    MonetaryAmount invoiceBalance =  
        new MonetaryAmount(5.0, Currency.USD);  
    Invoice invoice = new Invoice(invoiceBalance, "test-id");  
    PaymentManager paymentManager = new PaymentManager();  
  
    paymentManager.settleInvoice(mockAccount, invoice);  
    verifyThat(mockAccount.debit)  
        .wasCalledOnce()  
        .withArguments(invoiceBalance);  
}
```

BankAccount의 목 객체가
생성된다.

테스트 대상 코드는 mockAccount를
인수로 해서 호출된다.

mockAccount.debit() 함수가
예상한 인수로 호출되는지 확인한다.

주요 테스트 사항:

1. createMock(BankAccount)로 BankAccount 인터페이스의 Mock 객체를 생성합니다.
2. 이 Mock 객체를 settleInvoice() 메서드에 전달하여 테스트 대상 코드에 전달합니다.

3. 테스트에서 mockAccount.debit()가 송장 잔액으로 한 번 호출되었는지 확인하여, 정확한 금액이 인출되었음을 검증합니다.

이렇게 하면 테스트가 외부 시스템에 영향을 주지 않도록 보호할 수 있다는 이점이 있습니다. 하지만 반대로, Mock을 사용하는 방식에는 테스트가 실제 상황과 다르게 작동하거나 중요한 버그를 발견하지 못하는 위험이 있습니다. 이 부분은 10.4.4 절에서 살펴보겠습니다.

10.4.3 스텁

예제 10.5 getBalance()를 호출하는 코드

```
class PaymentManager {  
    ...  
  
    PaymentResult settleInvoice(  
        BankAccount customerBankAccount,  
        Invoice invoice) {  
        if (customerBankAccount.getBalance()  
            .isLessThan(invoice.getBalance())) {  
            return PaymentResult.insufficientFunds(invoice.getId());  
        }  
        customerBankAccount.debit(invoice.getBalance());  
        return PaymentResult.paid(invoice.getId());  
    }  
}
```

코드는 customerBankAccount.getBalance()가 반환하는 값에 의존한다.

Stub은 특정 함수가 호출되면 사전에 정해진 값을 반환하는 방식으로 함수를 시뮬레이션합니다. 이를 통해 테스트 대상 코드가 의존하는 함수 호출에서 특정 값을 얻도록 시뮬레이션할 수 있습니다. 따라서 Stub은 테스트 대상 코드가 의존 코드로부터 특정한 값을 필요로 할 때 유용하게 사용됩니다.

Mock과 Stub은 역할에 차이가 있지만, 개발자들은 일상적으로 두 용어를 혼용해서 사용하기도 합니다. 또한, 많은 테스트 도구는 Stub 기능만 필요해도 Mock을 생성하게 합니다.

예를 들어, PaymentManager.settleInvoice() 함수가 인출 전에 계좌 잔액이 충분한지 확인하는 방식으로 수정되었다고 가정해 봅시다. 이렇게 수정함으로써 거래가 거절되는 경우를 줄이고 고객의 신용 등급에 부정적인 영향을 미치지 않도록 도울 수 있지만, 새로운 기능으로 인해 테스트 케이스를 추가해야 하는 동작이 더 많아졌습니다.

1. 잔액이 부족할 경우 PaymentResult가 “잔액 부족”을 반환해야 합니다.
2. 잔액이 부족할 경우 실제로 계좌에서 인출이 이루어지지 않아야 합니다.
3. 잔액이 충분할 때 계좌에서 인출이 이루어져야 합니다.

이처럼 계좌 잔액에 따라 테스트 케이스가 달라져야 합니다. 그러나 실제 BankAccountImpl을 사용하면 테스트 대상 코드가 실제 계좌 잔액을 읽게 되어, 자주 변동될 수 있는 값을 사용하는 경우 테스트가 비결정적이게 됩니다. 이런 상황에서는 외부 환경으로부터 테스트를 보호해야 하므로, BankAccount.getBalance()에 Stub을 사용하여 정해진 값을 반환하게 설정합니다. 이를 통해 코드가 올바로 동작하는지 검증하고, 테스트 결과 역시 결정적이고 신뢰할 수 있게 됩니다.

예제 10.6 스텁을 사용하는 테스트 케이스

```
void testSettleInvoice_insufficientFundsCorrectResultReturned() {  
    MonetaryAmount invoiceBalance =  
        new MonetaryAmount(10.0, Currency.USD);  
    Invoice invoice = new Invoice(invoiceBalance, "test-id");  
    BankAccount mockAccount = createMock(BankAccount); ←  
    when(mockAccount.getBalance())  
        .thenReturn(new MonetaryAmount(9.99, Currency.USD));  
    PaymentManager paymentManager = new PaymentManager();  
  
    PaymentResult result =  
        paymentManager.settleInvoice(mockAccount, invoice);  
  
    assertThat(result.getStatus()).isEqualTo(INSUFFICIENT_FUNDS); ←  
}
```

스텁만 필요하지만
BankAccount 인터페이스에 대한
목 객체를 생성한다.

mockAccount.getBalance() 함수는
스텁을 통해 항상 9.99달러를 반환하도록
설정된다.

‘잔액 부족’이라는
결과가 반환되는지
확인한다.

Stub을 사용하면 테스트를 외부 환경으로부터 보호하고 결과에 신뢰성을 부여할 수 있습니다.

이번 절과 이전 절에서는 Mock과 Stub을 활용하여 의존성을 시뮬레이션하고, 테스트를 격리하는 방법이 어떻게 도움이 되는지 다루었습니다. 이렇게 격리하지 않으면 다양한 문제가 발생할 수 있습니다.

하지만 Mock과 Stub 사용에도 몇 가지 단점이 존재합니다. 다음 절에서는 이러한 주요 단점 두 가지를 설명합니다.

10.4.4 목과 스텁은 문제가 될 수 있다

Mock과 Stub을 사용할 때는 두 가지 주요 단점이 있습니다.

- 첫째, Mock이나 Stub이 실제 의존성과 다르게 동작하도록 설정되면 테스트가 실제 상황을 반영하지 않아 신뢰성이 떨어질 수 있습니다.
- 둘째, 구현 세부 사항과 테스트가 밀접하게 결합되어 리팩토링이 어려워질 수 있습니다.

목과 스텁은 실제적이지 않은 테스트를 만들 수 있다.

예제 10.7 마이너스 송장 잔액 테스트

```
void testSettleInvoice_negativeInvoiceBalance() {  
    BankAccount mockAccount = createMock(BankAccount);  
    MonetaryAmount invoiceBalance =  
        new MonetaryAmount(-5.0, Currency.USD); ← 마이너스 송장 잔액  
    Invoice invoice = new Invoice(invoiceBalance, "test-id");  
    PaymentManager paymentManager = new PaymentManager();  
  
    paymentManager.settleInvoice(mockAccount, invoice);  
  
    verifyThat(mockAccount.debit)  
        .wasCalledOnce()  
        .withArguments(invoiceBalance);  
}
```

테스트는 `mockAccount.debit()` 함수가 음수값으로 호출되는 것을 확인한다.

`PaymentManager.settleInvoice()` 함수의 테스트에서는 고객이 지불할 송장 잔액이 -5달러일 때를 가정하여 Mock을 사용해 진행했습니다. 여기서 -5달러라는 음수 잔액은 고객이 환불이나 보상을 받은 상황을 고려한 것입니다. 이러한 상황도 처리할 수 있도록 테스트가 필요합니다.

테스트 케이스에서는 `mockAccount.debit()`가 마이너스 잔액으로 호출되는지만 확인했는데, 실제 `BankAccountImpl.debit()` 함수가 음수 값으로 호출될 때는 실제 환경에서 문제가 발생할 가능성이 큽니다. 예를 들어, **PaymentManager** 클래스가 마이너스 금액을 인출(debit)하면 계좌에 돈이 추가된다는 가정을 하고 있지만, 이 가정이 실제로 테스트된 적이 없다는 것입니다.

```
interface BankAccount {  
    /**  
     * @throws ArgumentException 0보다 적은 금액으로 호출되는 경우  
     */  
    void debit(MonetaryAmount amount);  
  
    /**  
     * @throws ArgumentException 0보다 적은 금액으로 호출되는 경우  
     */  
    void credit(MonetaryAmount amount);  
}
```

또한 **BankAccount** 인터페이스에 대한 주석을 보면, 음수 값이 전달될 경우 **ArgumentException**이 발생한다고 명시되어 있을 수 있습니다. 하지만 Mock을 사용한 테스트에서는 이러한 조건을 고려하지 않고 테

스트를 통과시키기 때문에, 실제 상황에서 오류가 드러나지 않게 됩니다. 이는 Mock과 Stub의 주요 단점 중 하나로, 테스트 작성자가 **Mock** 또는 **Stub**이 실제 의존성처럼 동작할 것이라는 가정을 잘못 설정하게 되면, 현실과 동떨어진 테스트 결과가 나올 수 있습니다.

```
interface BankAccount {  
    ...  
  
    /**  
     * @return 가장 가까운 10의 배수로 반내림한 계좌의 잔액  
     * 예를 들어 실제 잔액이 19달러라면 이 함수는 10달러를 반환한다.  
     * 이것은 보안을 위한 것인데 정확한 잔액은 보안 확인을 위한 질문으로  
     * 은행이 사용하기 때문이다.  
     */  
    MonetaryAmount getBalance();  
}
```

NOTE | 잔액 반내림

반내림한 값을 반환하는 `getBalance()`의 예는 함수를 스텝할 때 특정 세부 사항을 간과하기가 얼마나 쉬운지 보여준다. 실제로 계좌 잔고를 반내림한다고 해서 특별하게 강력한 보안이 되는 것은 아니다. `getBalance()` 함수가 반환하는 값이 바뀔 때까지 계속해서 \$0.01씩 계좌이체를 하면 잔액을 파악할 수 있기 때문이다.

스텝을 사용할 때도 문제가 발생할 수 있습니다. 스텝을 통해 특정 값을 반환하도록 의존성 코드를 시뮬레이션하여, 테스트 대상 코드가 예상대로 동작하는지를 확인합니다. 하지만 이 반환 값이 실제 의존성 코드가 반환하는 값과 일치하는지는 확인하지 않습니다.

예를 들어, `BankAccount.getBalance()` 함수를 스텝으로 설정하여 특정 잔액을 반환하도록 시뮬레이션 할 수 있습니다. 그러나 이 함수의 실제 계약 조건, 즉 함수가 어떤 상황에서 어떤 값을 반환해야 하는지를 충분히 고려하지 않았을 수 있습니다. 만약 `BankAccount` 인터페이스의 문서를 자세히 검토했다면, 스텝이 실제 계약 조건을 반영하지 못한 부분을 발견할 수도 있었습니다.

결과적으로, 스텝을 사용할 때 실제 동작과 테스트 설정 사이에 차이가 발생할 수 있으며, 이로 인해 테스트가 부정확한 가정을 기반으로 이루어질 위험이 있습니다.

목과 스텝을 사용하면 테스트가 구현 세부 정보에 유착될 수 있다.

```
PaymentResult settleInvoice(...) {  
    ...  
    MonetaryAmount balance = invoice.getBalance();  
    if (balance.isPositive()) {  
        customerBankAccount.debit(balance);  
    } else {  
        customerBankAccount.credit(balance.absoluteAmount());  
    }  
    ...  
}
```

PaymentManager.settleInvoice() 함수가 마이너스 잔액이 있는 경우 credit()을, 양수인 경우 debit()을 호출하도록 개선한 코드이다.

```
void testSettleInvoice_positiveInvoiceBalance() {  
    ...  
    verifyThat(mockAccount.debit)  
        .wasCalledOnce()  
        .withArguments(invoiceBalance);  
}
```

```
void testSettleInvoice_negativeInvoiceBalance() {  
    ...  
    verifyThat(mockAccount.credit)  
        .wasCalledOnce()  
        .withArguments(invoiceBalance.absoluteAmount());  
}
```

이 코드를 목을 사용하여 테스트하면 예상되는 함수 호출이 발생했는지만 확인하게 됩니다. 즉, debit()이나

`credit()`이 호출되었는지만 검증할 수 있습니다. 하지만 실제 중요한 동작, 즉 정확한 금액이 계좌에 반영되는지 여부는 직접적으로 테스트하지 않습니다. 이 경우 함수 호출 여부는 단지 구현 세부 사항일 뿐이며, 핵심은 송금 동작이 제대로 수행되는지입니다.

```
interface BankAccount {  
    ...  
  
    /**  
     * 지정된 금액을 계좌로 송금한다. 금액이 0보다 적으면  
     * 계좌로부터 인출하는 효과를 갖는다.  
     */  
    void transfer(MonetaryAmount amount);  
}
```

```
PaymentResult settleInvoice(...) {  
    ...  
    MonetaryAmount balance = invoice.getBalance();  
    customerBankAccount.transfer(balance.negate());  
    ...  
}
```

또한, 이 코드를 리팩터링하여 `BankAccountImpl` 클래스에 `transfer()`라는 새로운 함수를 추가하고 `settleInvoice()` 가 이 함수를 호출하도록 변경할 수 있습니다. 이 경우, `debit()` 과 `credit()` 을 호출하는 대신 `transfer()` 만 호출되므로, 목을 기반으로 작성된 기존 테스트는 실패하게 됩니다.

이는 구현 세부 사항을 확인하는 테스트에 의존하게 되는 문제를 보여줍니다. 테스트가 구현 세부 사항에 종속되면 리팩터링이 동작을 변경하지 않았더라도 테스트 케이스 수정을 강요받게 됩니다.

따라서 목과 스텝 사용에는 한계가 있으며, 가능하다면 실제 의존성이나 페이크(**fakes**)를 사용하는 것이 바람직합니다.

10.4.5 페이크

페이크(Fake)는 클래스나 인터페이스의 실제 동작을 단순화된 방식으로 시뮬레이션하는 대체 구현체로, 테스트에서 안전하게 사용할 수 있습니다.

페이크는 외부 시스템과 통신하지 않고 내부 멤버 변수를 통해 상태를 관리합니다. 중요한 점은 페이크가 실제 구현의 코딩 규약과 동일하게 동작해야 하므로, 페이크도 유지보수 대상이 됩니다. 즉, 실제 구현이 변경되면 페이크도 함께 수정되어야 합니다.

예제 10.8 페이크 BankAccount

```
class FakeBankAccount implements BankAccount {  
    private MonetaryAmount balance; ← BankAccount 인터페이스를  
    ← 멤버 변수를 통해 구현한다.  
    상태를 추적한다.  
    FakeBankAccount(MonetaryAmount startingBalance) {  
        this.balance = startingBalance;  
    }  
  
    override void debit(MonetaryAmount amount) {  
        if (amount.isNegative()) {  
            throw new ArgumentException("액수는 0보다 적을 수 없음");  
        } ← 액수가 음수이면 ArgumentException을  
        balance = balance.subtract(amount); 발생시킨다.  
    }  
  
    override void credit(MonetaryAmount amount) {  
        if (amount.isNegative()) {  
            throw new ArgumentException("액수는 0보다 적을 수 없음");  
        } ← 액수가 음수이면 ArgumentException을  
        balance = balance.add(amount); 발생시킨다.  
    }  
  
    override void transfer(MonetaryAmount amount) {  
        balance.add(amount);  
    }  
  
    override MonetaryAmount getBalance() {  
        return roundDownToNearest10(balance); ← 잔액은 가장 가까운 10의  
        ← 배수로 반내림해서 반환한다.  
    }  
  
    MonetaryAmount getActualBalance() {  
        return balance; ← 테스트에서 반내림되지 않은  
    } ← 정확한 잔액을 확인할 수 있는  
} ← 추가적인 함수
```

다음은 BankAccount 인터페이스의 페이크 구현 FakeBankAccount를 통한 주요 특징입니다:

1. BankAccount 구현: FakeBankAccount는 BankAccount 인터페이스를 구현하여 BankAccount가 필요한 모든 코드에서 사용할 수 있습니다.
2. 내부 상태 관리: 은행 백엔드 시스템과의 통신 대신, 멤버 변수를 통해 계좌 잔액을 추적합니다.
3. 코드 규약 강제: debit()나 credit() 메서드가 음수 값으로 호출되면 ArgumentException 예외를 발생시킵니다. 이를 통해 실제 BankAccount 구현과 동일한 방식으로 코드 규약을 강제하고, 잘못된 호출을 방지합니다.
4. 잔액 반올림 처리: getBalance()는 실제 구현과 동일하게 잔액을 가장 가까운 10의 배수로 반올림하여 반환합니다. 이 동작을 통해 예상치 못한 버그를 더 잘 찾아낼 수 있습니다.
5. 추가 기능 제공: 테스트를 위한 getActualBalance() 메서드를 추가하여, 반올림 없이 실제 잔액을 확인할 수 있습니다. 이는 getBalance()가 반올림한 잔액을 반환하기 때문에 테스트에서 실제 상태를 정확하게 확인해야 할 때 중요합니다.

페이크로 인해 보다 실질적인 테스트가 이루어 질 수 있다.

예제 10.9 마이너스 송장 잔액에 대해 페이크를 사용한 테스트

```
void testSettleInvoice_negativeInvoiceBalance() {
    FakeBankAccount fakeAccount = new FakeBankAccount(
        new MonetaryAmount(100.0, Currency.USD));           ] 100달러 잔액으로 초기화되어 생성된
    MonetaryAmount invoiceBalance =                         페이크 계좌
        new MonetaryAmount(-5.0, Currency.USD);           ← -5달러인 송장 잔액
    Invoice invoice = new Invoice(invoiceBalance, "test-id");
    PaymentManager paymentManager = new PaymentManager();
    paymentManager.settleInvoice(fakeAccount, invoice);   ← fakeAccount로 호출되는
                                                          테스트 대상 코드

    assertThat(fakeAccount.getActualBalance())
        .isEqualTo(new MonetaryAmount(105.0, Currency.USD)); ] 새로운 계좌 잔액이
                                                               105달러인지 확인한다.
}
```

이전 절에서 살펴본 마이너스 송장 잔액을 테스트하는 예시에서는, PaymentManager.settleInvoice() 메서드가 제대로 작동하는지 확인하기 위해 BankAccount.debit() 메서드에 대한 목(Mock)을 사용했습니다. 그러나 목 객체는 마이너스 금액을 허용했기 때문에, 실제로 debit() 메서드가 마이너스 금액을 허용하지 않는다는 점을 간과하고도 테스트가 통과되었습니다. 따라서 코드에 버그가 있었음에도 테스트는 이를 탐지하지 못했습니다.

만약 목 대신 페이크(Fake)를 사용했다면 이 버그를 쉽게 발견할 수 있었을 것입니다. FakeBankAccount를 사용하여 마이너스 금액 테스트 케이스를 작성하면, **PaymentManager.settleInvoice()**가 호출될 때 **FakeBankAccount.debit()** 메서드가 예외를 발생시키고, 테스트는 실패하게 됩니다. 이를 통해 코드에 버그가 있음을 즉시 알 수 있고, 코드 베이스에 병합하기 전 수정이 가능해집니다.

페이크를 사용하면 구현 세부 정보로부터 테스트를 분리할 수 있다.

```
...
    assertThat(fakeAccount.getActualBalance())
        .isEqualTo(new MonetaryAmount(105.0, Currency.USD));
```

목이나 스텁 대신 페이크(Fake)를 사용하는 주요 장점은, 테스트가 구현 세부 사항에 밀접하게 결합되지 않는다는 점입니다. 앞에서 살펴본 예시에서는 개발자가 코드를 리팩터링할 때 목(Mock)을 사용한 테스트가 실패할 가능성을 봤습니다. 이 경우, 테스트는 debit()이나 credit() 메서드가 호출되는지 확인하는 데 중점을 두었는데, 이는 구현 세부 사항에 해당합니다. 반대로 페이크를 사용한 경우, 최종 계좌 잔액이 올바른지 확인하므로 구현 세부 사항보다는 결과에 집중하게 됩니다.

페이크를 사용하면 코드는 의도한 기능을 통해 계좌에 입출금을 할 수 있고, 최종 결과가 동일하기만 하면 테스트는 통과합니다. 따라서 테스트는 구현 세부 사항에서 더 독립적이며, 리팩토링이 동작을 변경하지 않는 한 테스트는 실패하지 않습니다.

모든 의존성 코드가 자체 페이크를 갖고 있지는 않습니다. 페이크를 만들지 여부는 해당 코드의 관리 주체와 유지보수 의지에 달려 있습니다. 만약 팀이 특정 클래스나 인터페이스를 관리하고 있으며 실제 코드를 테스트에 사용하는 것이 적합하지 않다면 페이크를 구현하는 것이 좋습니다. 이를 통해 테스트 품질이 향상될 뿐 아니라, 의존하는 다른 개발자들에게도 도움이 됩니다.

실제 의존성을 테스트에서 사용할 수 없는 경우 테스트 더블을 사용하는 것이 필요합니다. 페이크가 존재한다면, 목이나 스텁보다 페이크를 사용하는 것이 더 나은 선택일 수 있습니다.

10.4.6 목에 대한 의견

단위 테스트에서 목(Mock)과 스텁(Stub)을 사용하는 방식에는 크게 두 가지 의견이 있습니다.

1. **목 지지자(Mockist/London 학파)**: 의존성을 실제로 사용하는 대신, 단위 테스트에서 반드시 목을 사용해야 한다고 주장합니다. 이들은 테스트 대상 코드가 의존성 코드와 상호작용하는지를 확인하기 위해 목을 사용하며, 특정 값을 반환받는 경우에는 스텁을 활용합니다.
2. **고전주의자(Classicist/Detroit 학파)**: 목과 스텁은 최소한으로 사용해야 하며, 가능한 한 실제 의존성을 사용해야 한다고 주장합니다. 실제 의존성 사용이 불가능한 경우에만 페이크(Fake)를 사용하는 것을 선호하며, 목과 스텁은 마지막 수단으로 고려해야 한다고 봅니다.

주요 차이점은 목 접근법은 코드가 어떤 상호작용을 하는지 테스트하고, 고전적 접근법은 최종 결과 상태와 의존성의 결과에 중점을 둡니다. 즉, 목은 코드가 “어떻게” 작동하는지 확인하는 반면, 고전적 접근법은 코드 실행의 “결과”에 집중합니다.

결론

필자는 초기에는 목 접근법을 사용했으나, 동작을 제대로 테스트하지 않고 리팩터링을 어렵게 만들었다고 느껴 고전적 접근법을 선호하게 되었습니다. 다만, 목 접근법이 더 나은 경우도 있을 수 있으므로, 각각의 방식의 장단점을 이해하고 필요에 따라 선택하는 것이 중요합니다.

느낀점

Mock에 대해서는 알고 있었는데 Stub과 Fake 방식은 처음 알게 되었습니다. 스프링 개발시 FakeRepository를 이용해 테스트 코드를 작성하는 것을 본 적이 있는데, 그때는 왜 그렇게 이용했는지 이해하지 못했지만 지금에서야 이해가 됩니다.

<https://velog.io/@swager253/Yunny-Bucks.-Test-Fake-Repository>

관심있으신 분들은 위 아티클을 한번 읽어보는 것을 추천드립니다!

10.5 테스트 철학으로 부터 신중하게 선택하라

테스트 철학과 방법론에는 다양한 접근이 있으며, 개발자들은 각 철학에서 자신에게 맞는 방식을 선택할 수 있습니다. 대표적인 테스트 철학에는 TDD, BDD, ATDD 등이 있으며, 각각의 주요 특징은 다음과 같습니다:

- 테스트 주도 개발 (TDD):** TDD는 실제 코드를 작성하기 전에 테스트 코드를 먼저 작성하는 방식을 지지합니다. 최소한의 코드로 테스트를 통과한 후, 리팩터링을 통해 구조를 개선하고 중복을 제거합니다. TDD는 테스트 격리, 단일 동작 테스트, 구현 세부 사항에 대한 테스트 회피 등을 강조합니다.
- 행동 주도 개발 (BDD):** BDD는 사용자, 고객, 비즈니스의 관점에서 소프트웨어가 수행해야 할 행동과 기능에 집중합니다. 원하는 행동은 이해하기 쉬운 형식으로 기록되며, BDD 테스트는 소프트웨어의 내부 속성보다는 외부에서 관찰 가능한 행동에 초점을 맞춥니다. 구체적인 행동 정의 방식은 조직마다 다를 수 있습니다.
- 수용 테스트 주도 개발 (ATDD):** ATDD는 고객의 관점에서 소프트웨어의 전반적인 동작을 확인하기 위해 자동화된 수용 테스트를 작성합니다. 이 테스트는 TDD와 마찬가지로 코드 구현 이전에 작성되며, 모든 수용 테스트를 통과하면 소프트웨어는 고객이 수용할 준비가 된 상태임을 의미합니다. ATDD와 BDD는 정의에 따라 겹치거나 비슷하게 보일 수 있습니다.

결론적으로, 테스트 철학과 방법론은 개발자들이 효과적이라 생각하는 방식의 문서화된 형태일 뿐, 궁극적으로는 고품질의 테스트 코드와 소프트웨어를 만드는 것이 목표입니다. 특정 철학이나 방법론을 문자 그대로 따르기보다는, 상황에 맞게 효과적인 방식을 선택하는 것이 중요합니다.

요약

- 모든 실제 코드에는 해당하는 단위 테스트가 포함되어야 합니다.
- 코드의 모든 동작을 검증하는 테스트 케이스를 작성해야 하며, given/when/then으로 요즘 많이 사용하는 것 같습니다.
- 좋은 단위 테스트의 주요 특징은 다음과 같습니다
 - 문제 코드를 정확히 탐지
 - 구현 세부 사항에 구애받지 않음
 - 실패 시 명확한 설명 제공
 - 이해하기 쉬운 테스트 코드
 - 빠르고 쉽게 실행 가능

- 테스트 더블은 실제 의존성을 사용할 수 없거나 현실적으로 어려운 경우 단위 테스트에 사용됩니다. 종류로는 **목(mock)**, **스텁(stub)**, **페이크(fake)** 가 있습니다.
- 목과 스텁을 사용한 테스트는 비현실적일 수 있고, 구현 세부 사항과 밀접하게 연관될 위험이 있습니다.
- 필자 의견:
 - 가능한 한 실제 의존성을 테스트에 사용하는 것이 좋습니다.
 - 불가능할 경우 페이크가 차선책이며, 목과 스텁은 최후의 수단으로만 사용해야 합니다.