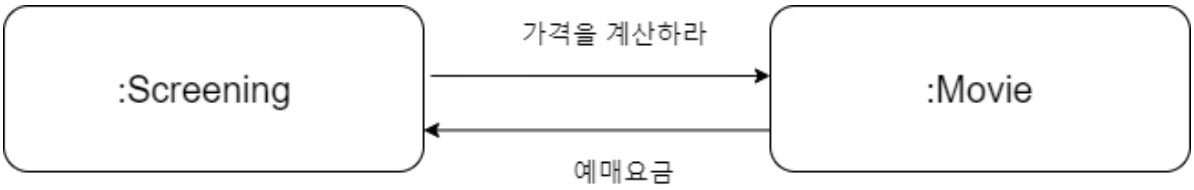
오브젝트 6장

협력과 메시지

> 협력은 어떤 객체가 다른객체에게 무언가를 요청할 때 시작한다.
>
> 메시지는 객체 사이의 협력을 가능하게 하는 매개체다.

두 객체 사이의 협력 관계를 설명하기 위해 사용하는 전통적인 메타포는 **클라이언트-서버(Client-Server)**모델이다 . 협력 안에서 메시지를 전송하는 객체를 클라이언트, 메시지를 수신하는 객체를 서버라고 부른다. 협력은 클라이언트가 서버의 서비스를 요청하는 단방향 상호작용이다.



다음은 영화 예매 시스템 예제에서 Screening 과 Movie 사이의 협력을 나타낸 것이다. Screening은 클라이언트의 역할을 수행하고 Movie는 서버의 역할을 수행한다. 클라이언트는 가격을 계산하라 메시지를 전송함으로써 도움을 요청하고 서버인 Movie는 가격을 계산하는 서비스를 제공함으로써 메시지에 응답한다.

- 요점은 객체가 독립적으로 수행할 수 있는 것보다 더 큰 책임을 수행하기 위해서는 다른객체와 협력해야 한다는 것이다. 그리고 두 객체 사이의 협력을 가능하게 해주는 매개체가 바로 메시지라는 것이다.

메시지와 메시지 전송

메시지는 객체들이 협력하기 위해 사용할 수 있는 유일한 의사소통 수단이다. 메시지 전송은 한 객체가 다른객체에게 도움을 요청하는 것인데, 메시지를 전송하는 객체를 메시지 전송자라고 하고 메시지를 수신하는 객체를 메시지 수신자라고 부른다.

클라이언트- 서버 모델에서는 메시지 전송자는 클라이언트, 메시지 수신자는 서버라고 부르기도 한다.

메시지는 **오퍼레이션명**과 **인자**로 구성되며 메시지 전송은 여기에 **메시지 수신자**를 추가한 것이다. 따라서 메시지 전송은 메시지 수신자, 오퍼레이션명, 인자의 조합이다.

메시지 전송의 표기법은 프로그래밍 언어에 따라 다르겠지만 메시지 전송을 구성하는 요소들은 동일하다. 자바로 예를 들어서 오퍼레이션 명과 인자를 조합한 isSatisfiedBy(screening)이 '메시지'이고, 여기에 메시지 수신자인 condition을 추가한 condition.isSatisfiedBy(screening)이 '메시지 전송'이다.

메시지와 메서드

메시지를 수신했을 때 실제로 어떤 코드가 실행되는지는 메시지 수신자의 실제 타입이 무엇인가에 달려 있다.

condition.isSatisfiedBy(screening) 이라는 메시지 전송 구문에서 메시지 수진자인 condition은 DiscountCondition 이라는 인터페이스 타입으로 정의돼 있지만 실제로 실행되는 코드는 인터페이스를 실체화한 클래스의 종류에 따라 달라진다. condition이 SequenceCondition의 인스턴스라면 SequenceCondition에 구현된 isSatisifiedBy 메서드가 실행될 것이다.

이처럼 메시지를 수신했을 때 실제로 실행되는 함수 또는 프로시저를 **메서드**라고 부른다. 중요한 것은 코드 상에서 동일한 이름의 변수에게 동일한 메시지를 전송하다라도 객체의 타입에 따라 실행되는 메서드가 달라질 수 있다는 것이다.

기술적인 관점에서 객체 사이의 메시지 전송은 전통적인 방식의 함수 호출이나 프로시저 호출과는 다르다. 전통적인 방식은 컴파일 시점과 실행시점의 메서드가 같은 반면 객체지향에서는 메시지와 메서드라는 두가지 서로 다른 개념을 실행 시점에 연결해야 하기 때문에 컴파일 시점과 실행 시점의 의미가 달라질 수 있다.

객체지향이 메시지 전송과 메서드 호출을 명확하게 구분한다는 사실이 여러분을 모호함의 덫으로 밀어넣을 수 있다. 메시지 전송을 코드상에 표기하는 시점에는 어떤 코드가 실행될 것인지를 정확하게 알 수 없다.

실행 시점에 실제로 실행되는 코드는 메시지를 수신하는 객체의 타입에 따라 달라지기 때문에 우리는 그저 메시지에 응답할 수 있는 객체가 존재하고, 그 객체가 적절한 메서드를 선택해서 응답할 것이라고 믿을 수 밖에 없다.

- 메시지와 메서드의 구분은 메시지 전송자와 메시지 수신자가 느슨하게 결합 될 수 있게 한다. 메시지 전송자와 메시지 수신자는 서로에 대한 상세한 정보를 알지 못한 채 단지 메시지라는 얇고 가는 끈을 통해 연결된다. 실행 시점에 메시지와 메서드를 바인딩하는 메커니즘은 두 객체 사이의 결합도를 낮춤으로써 유연하고 확장 가능한 코드를 작성할 수 있게 만든다.

퍼블릭 인터페이스와 오퍼레이션

객체는 안과 밖을 구분하는 뚜렷한 경계를 가진다. 외부의 객체는 오직 객체가 공개하는 메시지를 통해서만 객체와 상호작용할 수 있다. 이처럼 객체가 의사소통을 위해 외부로 공개하는 메시지의 집합을 **퍼블릭 인터페이스**라고 부른다.

프로그래밍 언어의 관점에서 퍼블릭 인터페이스에 포함된 메시지를 **오퍼레이션**이라고 부른다. 오퍼레이션은 수행 가능한 어떤 행동에 대한 추상화다.

그에 비에 메시지를 수신했을 때 실제로 실행되는 코드는 메서드라고 부른다.

> UML은 공식적으로 오퍼레이션을 다음과 같이 정의한다.
>
> 오퍼레이션이란 실행하기 위해 객체가 호출될 수 있는 변환이나 정의에 관한 명세다.
>
> UML 용어로 말하자면, 인터페이스의 각 요소는 오퍼레이션이다. 오퍼레이션은 구현이 아닌 추상화다, 반면 UML의 메서드는 오퍼레이션을 구현한 것이다. 인용하면 메서드는 오퍼레이션에 대한 구현이다. 메서드는 오퍼레이션과 연관된 알고리즘 또는 절차를 명시한다.

프로그래밍 언어의 관점에서 객체가 다른객체에게 메시지를 전송하면 런타임 시스템은 메시지 전송을 오퍼레이션 호출로 해석하고 메시지를 수진한 객체의 실제 타입을 기반으로 적절한 메서드를 찾아 실행한다. 따라서 퍼블릭 인터페이스와 메시지의 관점에서 보면 '메서드 호출' 보다는 '오퍼레이션 호출' 이라는 용어를 사용하는 것이 더 적절하다.

시그니처

오퍼레이션의 이름과 파라미터 목록을 합쳐 시그니처라고 부른다. 오퍼레이션은 실행 코드 없이 시그니처만을 정의한 것이다. 일반적으로 메시지를 수신하면 오퍼레이션의 시그니처와 동일한 메서드가 실행된다.

- 하나의 오퍼레이션에 대해 오직 하나의 메서드만 존재하는 경우 세상은 꽤나 단순해 진다. 이런 경우에는 굳이 오퍼레이션과 메서드를 구분하 ㄹ필요가 없다. 하지만 다형성의 축복을 받기 위해서는 하나의 오퍼레이션에 대해 다양한 메서드를 구현해야 한다. 따라서 오퍼레이션의 관점에서 다형성이란 동일한 오퍼레이션 호출에 대해 서로 다른메서드들이 실행되는 것이라고 정의할 수 있다.

용어 정리

- 메시지: 객체가 다른객체와 협력하기 위해 사용하는 의사소통 메커니즘, 일반적으로 객체의 오퍼레이션이 실행되도록 요청하는 것을 "메시지 전송" 이라고 부른다. 메시지는 협력에 함여하는 전송자와 수신자 양쪽 모두를 포함하는 개념이다.
- 오퍼레이션 : 객체가 다른 객체에게 제공하는 추상적인 서비스다. 메시지가 전송자와 수신자 사이의 협력 관계를 강조하는 데 비해 오퍼레이션은 메시지를 수신하는 객체의 인터페이스를 강조한다. 다시 말해서 메시지 전송자는 고려하지 않은 채 메시지 수신자의 관점만을 다룬다. 메시지 수신이란 메시지에 대응되는 객체의 오퍼레이션을 호출하는 것을 의미한다.
- 메서드 : 메시지에 응답하기 위해 실행되는 코드 블록을 메서드라고 부른다. 메서드는 오퍼레이션의 구현이다. 동일한 오퍼레이션이라고 해도 메서드는 다를 수 있다. 오퍼레이션과 메서드의 구분은 다형성의 개념과 연결된다.
- 퍼블릭 인터페이스 : 객체가 협력에 참여하기 위해 외부에서 수신할 수 있는 메시지의 묶음. 클래스의 퍼블릭 메서드들의 집합이나 메시지의 집합을 가리키는데 사용된다. 객체를 설계할 때 가장 중요한 것은 훌륭한 퍼블릭 인터페이스를 설계하는 것이다.
- 시그니처 : 시그니처는 오퍼레이션이나 메서드의 명세를 나타낸 것으로, 이름과 인자의 목록을 포함한다. 대부분의 언어는 시그니처의 일부로 반환 타입을 포함하지 않지만 반환 타입을 시그니처의 일부로 포함하는 언어도 존재한다.

인터페이스와 설계 품질

좋은 인터페이스는 최소한의 인터페이스와 추상적인 인터페이스라는 조건을 만족해야 한다.

최소주의를 따르면서도 추상적인 인터페이스를 설계할 수 있는 가장 좋은 방법은 책임 주도 설계 방법을 따르는 것이다. 책임 주도 설계 방법은 메시지를 먼저 선택함으로써 협력과는 무관한 오퍼레이션이 인터페이스에 스며드는 것을 방지한다. 따라서 인터페이스는 최소의 오퍼레이션만 포함하게 된다.

이번 장에서는 퍼블릭 인터페이스의 품질에 영향을 미치는 다음과 같은 원칙과 기법에 관해 살펴보겠다.

- 디미터 법칙
- 묻지 말고 시켜라
- 의도를 드러내는 인터페이스
- 명령 - 쿼리 분리

디미터 법칙

다음 코드는 4장에서 살펴본 절차적인 방식의 영화 예매 시스템 코드 중에서 할인 가능 여부를 체크하는 코드를 가져온 것이다.

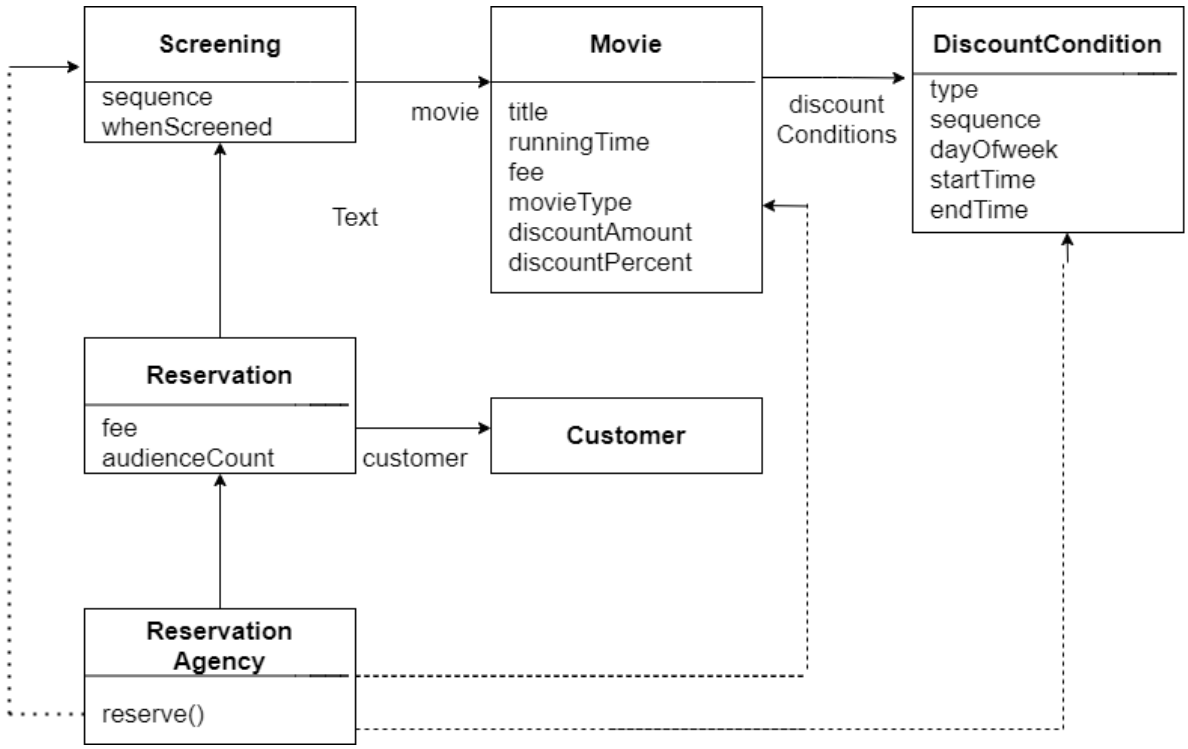
```
public class ReservationAgency {

    public Reservation reserve(Screening, Customer, int audienceCount) {

        Movie movie = screening.getMovie();

        boolean discountable = false;
        for (DiscountCondition conditon : movie.getDiscountConditions()) {
            if (condition.getType() == DiscountConditionType.PERIOD) {
                discountable =
screening.getWhenScreened().getDayOfWeek().equals(condition.getOfWeek()) &&
condition.getStartTime().compareTo(screening.getWhenScreened().toLocalTime()) <=
0 &&
condition.getEndTime().compareTo(screening.getWhenScreened().toLocalTime()) >=
0;
            } else {
                discountable = condition.getSequence() ==
screening.getSequence();
            }
        }
        if (discountable) {
            break;
        }
    }
}
```

이 코드의 단점은 ReservationAgency와 인자로 전달된 Screening 사이의 결합도가 너무 높기 때문에 Screening의 내부 구현을 변경할 때 마다 ReservationAgency도 함께 변경된다는 것이다. 문제의 원인은 ReservationAgency가 Screening 뿐만 아니라 Movie 와 DiscountCondition에도 직접 접근하기 때문이다.



screening 의 내부구조와 강하게 결합돼 있어 변경에 취약한 Reservation Agency

이처럼 협력하는 객체의 내부 구조에 대한 결합으로 인해 발생하는 설계 문제를 해결하기 위해 제안된 원칙이 바로 **디미터 법칙**이다.

디미터 법칙을 간단하게 요약하면 객체의 내부 구조에 강하게 결합되지 않도록 협력 경로를 제한하라는 것이다.

디미터의 법칙은 "낯선 자에게 말하지 말라" 또는 "오직 인접한 이웃하고만 말하라" 로 요약할 수 있다.

디미터의 법칙을 따르기 위해서는 클래스가 특정한 조건을 만족하는 대상에게만 메시지를 전송하도록 프로그래밍해야 한다. 모든 클래스 C 와 C에 구현된 모든 메서드 M에 대해서, M이 메시지를 전송할 수 있는 모든 객체는 다음에 서술된 클래스의 인스턴스여야 한다. 이때 M에 의해 생성된 객체나 M이 호출하는 메서드에 의해 생성된 객체, 전역변수로 선언된 객체는 모두 M의 인자로 간주한다.

- M의 인자로 전달된 클래스(C 자체를 포함)
- C의 인스턴스 변수의 클래스

즉 클래스 내부의 메서드가 아래 조건을 만족하는 인스턴스에만 메시지를 전송하도록 프로그래밍해야 한다라고 이해해도 무방하다.

- this 객체
- 메서드의 매개변수
- this의 속성
- this의 속성인 컬렉션의 요소
- 메서드 내에서 생성된 지역 객체

4장에서 결합도 문제를 해결하기 위해 수정한 ReservationAgency의 최종 코드를 보자

```
public class ReservationAgency {
    public Reservation reserve(Screening screening, Customer customer, int

audienceCount){
        Money fee = screening.calculateFee(audienceCount);
        return new Reservation(customer, screening, fee, audienceCount);
    }
}
```

이 코드에서 ReservationAgency는 메서드의 인자로 전달된 Screening 인스턴스에게만 메시지를 전송한다.

ReservationAgency는 Screening 내부에 대한 어떤 정보도 알지 못한다. ReservationAgency가 Screening의 내부 구조에 결합돼 있지 않기 때문에 Screening의 내부 구현을 변경할 때 ReservationAgency를 함께 변경할 필요가 없다.

디미터 법칙을 따르면 **부끄럼타는 코드(shy code)**를 작성할 수 있다. 부끄럼타는 코드란 불필요한 어떤 것도 다른객체에게 보여주지 않으며, 다른객체의 구현에 의존하지 않는 코드를 말한다. 디미터 법칙을 따르는 코드는 메시지 수신자의 내부 구조가 전송자에게 노출되지 않으며, 메시지 전송자는 수신자의 내부 구현에 결합되지 않는다.

> 디미터 법칙과 캡슐화
>
> 디미터 법칙은 캡슐화를 다른관점에서 표현한 것이다. 디미터 법칙이 가치 있는 이유는 클래스를 캡슐화하기 위해 따라야하는 구체적인 지침을 제공하기 때문이다. 캡슐화 원칙이 클래스 내부의 구현을 감춰야 한다는 사실을 강조하면 디미터 법칙은 협력하는 클래스의 캡슐화를 지키기 위해 접근해야 하는 요소를 제한한다.

다음은 디미터 법칙을 위반하는 코드의 전형적인 모습을 표현한 것이다.

```
screening.getMovie().getDiscountConditions();
```

메시지 전송자가 수신자의 내부 구조에 대해 물어보고 반환 받은 요소에 대해 연쇄적으로 메시지를 전송한다. 흔히 이와 같은 코드를 기차충돌이라고 부르는데 여러 대의 기차가 한 줄로 늘어서 충돌한 것처럼 보이기 때문이다. 기차 충돌은 클래스의 내부 구현이 외부로 노출됐을 때 나타나는 전형적인 형태로 메시지 전송자는 메시지 수신자의 내부 정보를 자세히 알게 된다.

다음은 디미터의 법칙을 따르는 코드다.

```
screening.calculateFee(audienceCount);
```

디미터 법칙은 객체가 자기 자신을 책임지는 자율적인 존재여야 한다는 사실을 강조한다. 정보를 처리하는 데 필요한 책임을 정보를 알고 있는 객체에게 할당하기 때문에 응집도가 높은 객체가 만들어진다.

디미터의 법칙을 무비판적으로 수용하면 퍼블릭 인터페이스 관점에서 객체의 응집도가 낮아질 수도 있다. 디미터의 법칙은 객체의 내부 구조를 묻는 메시지가 아니라 수신자에게 무언가를 시키는 메시지가 더 좋은 메시지라고 속삭인다.

묻지 말고 시켜라

디미터 법칙은 훌륭한 메시지는 객체의 상태에 관해 묻지 말고 원하는 것을 시켜야 한다는 사실을 강조한다.

묻지말고 시켜라는 이런 스타일의 메시지 작성을 장려하는 원칙을 가리키는 용어다.

메시지 전송자는 메시지 수신자의 상태를 기반으로 결정을 내린 후 메시지 수신자의 상태를 바꿔서는 안된다.

> 절차적인 코드는 정보를 얻은 후에 결정한다. 객체지향 코드는 객체에게 그것을 하도록 시킨다.

- 객체의 정보를 이용하는 행동을 객체의 외부가 아닌 내부에 위치시키기 때문에 자연스럽게 정보와 행동을 동일한 클래스 안에 두게 된다.
- 상태를 묻는 오퍼레이션을 행동을 요청하는 오퍼레이션으로 대체함으로써 인터페이스를 향상시켜라. 협력을 설계하고 객체가 수신할 메시지를 결정하는 매 순간 묻지말고 시켜라. 원칙과 디미터 법칙을 머릿속에 떠올리는 것은 퍼블릭 인터페이스의 품질을 향상 시킬 수 있는 좋은 습관이다.

하지만 단순하게 객체에게 묻지 않고 시킨다고 해서 모든 문제가 해결되는 것은 아니다. 훌륭한 인터페이스를 수확하기 위해서는 객체가 어떻게 작업을 수행하는지를 노출해서는 안된다. 인터페이스는 객체가 어떻게 하는지가 아니라 무엇을 하는지를 서술해야 한다.

의도를 드러내는 인터페이스

켄트 백은 그의 책에서 메서드를 명명하는 두가지 방법을 설명했다.

1. 메서드가 작업을 어떻게 수행하는지를 나타내도록 이름짓는 것이다.

   이 경우 메서드의 이름은 내부의 구현 방법을 드러낸다. 다음은 첫 번째 방법에 따라 PeriodCondition 과 SequenceCondition의 메서드를 명명한 것이다.

   ```
   public class PeriodCondition {
       public boolean isSatisfiedByPeriod(Screening screening) { ... }

   }

   public class SequenceCondition {
       public boolean isSatisfiedBySequence(Screening screening) { ... }
   }
   ```

   이런 스타일은 좋지 않은데 이유를 두가지로 요약할 수 있다.

   - 메서드에 대해 제대로 커뮤니케이션하지 못한다. 클라이언트 관점에서 두 메서드는 모두 할인 조건을 판단하는 동일한 작업을 수행한다. 하지만 메서드 이름이 다르기 때문에 두 메서드의 내부 구현을 정확하게 이해하지 못한다면 두 메서드가 동일한 작업을 수행한다는 사실을 알아채기 어렵다.
   - 더 큰 문제는 메서드 수준에서 캡슐화를 위반한다는 것이다. 이 메서드들은 클라이언트로 하여금 협력하는 객체의 종류를 알도록 강요한다. PeriodCondition을 사용하는 코드를 SequenceCondition을 사용하도록 변경하려면 단순히 참조하는 객체를 변경하는 것 뿐만 아니라 호출하는 메서드를 변경해야 한다. 만약 할인 여부를 판단하는 방법이 변경된다면 메서

드의 이름역시 변경해야 할 것이다. 메서드 이름을 변경한다는 것은 메시지를 전송하는 클라이언트의 코드도 함께 변경되야 한다는 것을 의미한다.

따라서 책임을 수행하는 방법을 드러내는 메서드를 사용한 설계는 변경에 취약할 수 밖에 없다.

2. 두 번째 방법은 "어떻게" 가 아니라 "무엇" 을 하는지를 드러내는 것이다. 메서드의 구현이 한 가지인 경우에는 무엇을 하는지를 드러내는 이름을 짓는 것이 어려울 수도 있다. 하지만 무엇을 하는지를 드러내는 이름은 코드를 읽고 이해하기 쉽게 만들 뿐만 아니라 유연한 코드를 낳는 지름길이다.
    - 어떻게 수행하는지를 드러내는 이름이란 메서드의 내부 구현을 설명하는 이름이다. 결과적으로 협력을 설계하기 시작하는 이른시기부터 클래스의 내부 구현에 관해 고민할 수 밖에 없다. 반면 무엇을 하는지를 드러내도록 메서드의 이름을 짓기 위해서는 객체가 협력 안에서 수행해야 하는 책임에 관해 고민해야 한다.

이제 무엇을 하는지를 드러내도록 isSatisfiedByPeriod와 isSatisfiedBySequence의 이름을 변경하자. 이 물음에 답하기 위해선 클라이언트의 관점에서 협력을 바라봐야 한다.

```
public class PeriodCondition {
    public boolean isSatisfiedBy(Screening screening) {...}
}

public class SequenceCondition {
    public boolean isSatisfiedBy(Screening screening) {...}
}
```

변경된 코드는 PeriodCondition의 isSatisfiedBy 메서드와 SequenceCondition의 isSatisfiedBy 메서드가 동일한 목적을 가진다는 것을 메서드의 이름을 통해 명확하게 표현한다.

클라이언트는 두메서드를 가진 객체를 동일한 타입으로 간주할 수 있도록 동일한 타입 계층으로 묶어야 한다. 가장 간단한 방법은 DiscountCondition이라는 인터페이스를 정의하고 이 인터페이스에 isSatisfiedBy 오퍼레이션을 정의하는 것이다.

```
public interface DiscountCondition {
    boolean isSatisfiedBy(Screening screening)
}
```

PeriodConditiond와 SequenceCondition을 동일한 타입으로 선언하기 위해 DiscountCondition 인터페이스를 실체화 하면 클라이언트 입장에서 두 메서드를 동일한 방식으로 사용할 수 있게 된다.

이제 무엇을 하느냐에 따라 메서드의 이름을 짓는 것이 설계를 유연하게 만드는 이유를 이해했을 것이다. 메서드가 어떻게 수행하느냐가 아니라 무엇을 하느냐에 초점을 맞추면 클라이언트 관점에서 동일한 작업을 수행하는 메서드들을 하나의 타입 계층으로 묶을 수 있는 가능성이 커진다.

이와 같은 메서드의 이름을 짓는 패턴을 **의도를 드러내는 선택자**라고 부른다. 켄트 벡은 메서드에 의도를 드러낼 수 있는 이름을 붙이기 위해 다음과 같이 조언한다.

> 하나의 구현을 가진 메시지의 이름을 일반화하도록 도와주는 간단한 훈련 방법을 소개하겠다. 매우 다른 두번째 구현을 상상하라. 그러고는 해당 메서드에 동일한 이름을 붙인다고 상상해보라. 그렇게 하면 아마도 그 순간에 여러분이 할 수 있는 한 가장 추상적인 이름을 메서드에 붙일 것이다.

도메인 주도 설계에서 에익 에반스는 켄트 백의 의도를 드러내는 선택자를 인터페이스 레벨로 확장한 의도를 드러내는 인터페이스를 제시했다. 구현과 관련된 모든 정보를 캡슐화하고 객체의 퍼블릭 인터페이스에는 협력과 관련된 의도만을 표현해야 한다는 것이다.

- 객체에게 묻지 말고 시키되 구현 방법이 아닌 클라이언트의 의도를 드러내야 한다. 이것이 이해하고 쉽고 유연한 동시에 협력적인 객체를 만드는 가장 기본적인 요구사항이다.

함께 모으기

근본적으로 디미터 법칙을 위반하는 설계는 **인터페이스의 구현의 분리 원칙**을 위반한다. 기억해야 할 점은 객체의 내부 구조는 구현에 해당한다는 것이다.

일반적으로 프로그램에 노출되는 객체 사이의 관계가 많아질수록 결합도가 높아지기 때문에 프로그램은 불안정해진다. 객체의 구조는 다양한 요구사항에 의해 변경되기 쉽기 때문에 디미터 법칙을 위반한 설계는 요구사항 변경에 취약해 진다.

디미터 법칙을 위반한 코드는 사용하기도 어렵다. 클라이언트 객체의 개발자는 퍼블릭 인터페이스 뿐만 아니라 내부 구조까지 속속들이 알고 있어야 하기 때문이다.

디미터 법칙을 위반한 코드를 수정하는 일반적인 방법은 내부구조를 묻는 대신 직접 자신의 책임을 수행하도록 시키는 것이다.

묻지 말고 시켜라

디미터 법칙과 묻지 말고 시켜라 원칙에 따라 코드를 리팩토링 한다면 객체가 스스로 자신의 상태를 제어하게 된다. 자신의 상태를 스스로 관리하고 결정하는 자율적인 존재가 된 것이다.

지금까지 살펴본 것처럼 디미터 법칙과 묻지 말고 시켜라 스타일을 따르면 자연스럽게 자율적인 객체로 구성된 유연한 협력을 얻게 된다. 구현이 객체의 퍼블릭 인터페이스에 노출되지 않기 때문에 객체 사이의 결합도는 낮아진다. 책임이 잘못된 곳에 할당될 가능성이 낮아지기 때문에 객체의 응집도 역시 높아진다. 일단 디미터 법칙과 묻지 말고 시켜라 스타일을 따르는 인터페이스를 얻었다면 인터페이스가 클라이언트의 의도를 올바르게 반영했는지를 확인해야 한다.

인터페이스에 의도를 드러내자

오퍼레이션의 이름은 협력이라는 문맥을 반영해야 한다. 오퍼레이션은 클라이언트가 객체에게 무엇을 원하는지를 표현해야 한다. 다시 말해 객체 자신이 아닌 클라이언트의 의도를 표현하는 이름을 가져야 한다.

디미터 법칙은 객체 간의 협력을 설계할 때 캡슐화를 위반하는 메시지가 인터페이스에 포함되지 않도록 제한한다. 묻지 말고 시켜라 원칙은 디미터 법칙을 준수하는 협력을 만들기 위한 스타일을 제시한다. 여기서 멈추지 않고, 의도를 드러내는 인터페이스 원칙은 객체의 퍼블릭 인터페이스에 어떤 이름이 드러나야 하는지에 대한 지침을 제공함으로써 코드의 목적을 명확하게 커뮤니케이션할 수 있게 해준다.

- 우리는 결합도가 낮으면서도 의도를 명확히 드러내는 간결한 협력을 원한다. 디미터 법칙과 묻지 말고 시켜라 스타일, 의도를 드러내는 인터페이스가 우리를 도울 것이다.

> 정보은닉 말고도 "묻지 말고 시켜라" 스타일에는 좀 미묘한 이점이 있다. 이 스타일은 객체 간의 상호작용을 getter의 체인 속에 암시적으로 두지 않고 좀 더 명시적으로 만들고 이름을 가지도록 강요한다.

원칙의 함정

디미터 법칙과 묻지 말고 시켜라 스타일은 객체의 퍼블릭 인터페이스를 깔끔하고 유연하게 만들 수 있는 훌륭한 설계 원칙이다. 하지만 절대적인 법칙은 아니다. 소프트웨어 설계에 법칙이란 존재하지 않는다. 법칙에는 예외가 없지만 원칙에는 예외가 넘쳐난다.

잊지 말아야 하는 사실은 설계가 트레이드오프의 산물이라는 것이다 설계를 적절하게 트레이드 오프 할 수 있는 능력이 숙련자와 초보자를 구분하는 가장 중요한 기준이라고 할 수 있다.

원칙이 현재 상황에 부적합하다고 판단된다면, 과감하게 원칙을 무시하라. 원칙을 하는 것보다 더 중요한 것은 언제 원칙이 유용하고 언제 유용하지 않은지를 판단할 수 있는 능력을 기르는 것이다.

디미터 법칙은 하나의 도트(.)를 강제하는 규칙이 아니다.

앞서 디미터 법칙은 "오직 하나의 도트만을 사용하라" 라는 말로 요약되기도 한다. 따라서 대부분의 사람들은 자바 8의 IntStream을 사용한 아래의 코드가 기차 충돌을 초래하기 때문에 디미터 법칙을 위반한다고 생각할 것이다.

```
IntStream.of(1, 15, 20, 3, 9).filter(x -> x > 10).distinct().count();
```

하지만 이것은 디미터 법칙을 제대로 이해하지 못한 것이다. 위 코드에서 of, filter, distinct 메서드는 모두 IntStream이라는 동일한 클래스의 인스턴스를 반환한다. 즉, 이들은 IntStream의 인스턴스를 또 다른 IntStream의 인스턴스로 변환한다.

따라서 이 코드는 디미터 법칙을 위반하지 않는다 디미터 법칙은 결합도와 관련된 것이며, 이 결합도가 문제가 되는 것은 객체의 내부 구조가 외부로 노출되는 경우로 한정된다. IntStream의 내부 구조가 외부로 노출되지 않았고, IntStream을 다른 IntStream으로 변환할 뿐, 객체를 둘러싸고 있는 캡슐은 그대로 유지된다.

하나 이상의 도트를 사용하는 모든 케이스가 디미터 법칙 위반인 것은 아니다. 기차 충돌처럼 보이는 코드라도 객체의 내부 구현에 대한 어떤 정보도 외부로 노출하지 않는다면 그것은 디미터 법칙을 준수한 것이다.

이 메서드들은 객체의 내부에 대한 어떤 내용도 묻지 않는다. 그저 객체를 다른객체로 변환하는 작업을 수행하라고 시킬 뿐이다. 따라서 묻지 말고 시켜라 원칙을 위반하지 않는다.

결합도와 응집도의 충돌

일반적으로 어떤 객체의 상태를 물어본 후 반환된 상태를 기반으로 결정을 내리고 그 결정에 따라 객체의 상태를 변경하는 코드는 묻지 말고 시켜라 스타일로 변경해야 한다.

안타깝게도 묻지 말고 시켜라와 디미터 법칙을 준수하는 것이 항상 긍정적인 결과로만 귀결되는 것은 아니다. 모든 상황에서 맹목적으로 위임 메서드를 추가하면 같은 퍼블릭 인터페이스 안에 어울리지 않는 오퍼레이션들이 공존하게 된다. 결과적으로 객체는 상관 없는 책임들을 한꺼번에 떠안게 되기 때문에 결과적으로 응집도가 낮아진다.

클래스는 하나의 변경 원인만을 가져야 한다. 서로 상관 없는 책임들이 함께 뭉쳐있는 클래스는 응집도가 낮으며 작은 변경으로도 쉽게 무너 질 수 있다. 따라서 디미터 법칙과 묻지 말고 시켜라 원칙을 무작정 따르면 애플리케이션은 응집도가 낮은 객체로 넘쳐날 것이다.

> 로버트 마틴은 클린코드에서 디미터 법칙의 위반 여부는 묻는 대상이 객체인지, 자료 구조인지에 달려있다고 설명한다. 객체는 내부 구조를 숨겨야 하므로 디미터 법칙을 따르는 것이 좋지만 자료 구조라면 당연히 내부를 노출해야 하므로 디미터 법칙을 적용할 필요가 없다.

- 객체에게 시키는 것이 항상 가능한 것은 아니다. 가끔씩은 물어야 한다. 여기서 강조하고 싶은 것은 소프트웨어 설계에 법칙이란 존재하지 않는 다는 것이다. 원칙을 맹신하지 마라. 원칙이 적절한 상황과 부적절한 상황을 판단할 수 있는 안목을 길러라. 설계는 트레이드오프의 산물이다.
- 소프트웨어 설계에 존재하는 몇 안 되는 법칙 중 하나는 "경우에 따라 다르다" 라는 사실을 명심하라.

명령-쿼리 분리 원칙

가끔씩은 필요에 따라 물어야 한다는 사실에 납득 했다면 명령-쿼리 분리 원칙을 알아두면 도움이 될 것이다.

명령-쿼리 분리 원칙은 퍼블릭 인터페이스에 오퍼레이션을 정의할 때 참고 할 수 있는 지침을 제공한다.

어떤 절차를 묶어 호출 가능하도록 이름을 부여한 기능 모듈을 **루틴**이라고 부른다. 루틴은 다시 **프로시저**와 **함수**로 구분할 수 있다. 프로시저와 함수를 같은 의미로 혼용하는 경우가 많지만 사실 프로시저와 함수는 부수효과와 반환값의 유무라는 측면에서 명확하게 구분된다.

- 프로시저는 정해진 절차에 따라 내부의 상태를 변경하는 루틴의 한 종류다.
- 이에 반해 함수는 어떤 절차에 따라 필요한 값을 계산해서 반환하는 루틴의 한 종류다.

프로시저와 함수를 명확하게 구분하기 위해 루틴을 작성할 때 다음과 같은 제약을 따라야 한다.

- 프로시저는 부수효과를 발생시킬 수 있지만 값을 반환할 수 없다.
- 함수는 값을 반환할 수 있지만 부수효과를 발생시킬 수 없다.

명령과 **쿼리**는 객체의 인터페이스 측면에서 프로시저와 함수를 부르는 또 다른이름이다. 객체의 상태를 수정하는 오퍼레이션을 명령이라고 부르고 객체와 관련된 정보를 반환하는 오퍼레이션을 쿼리라고 부른다. 따라서 개념적으로 명령은 프로시저와 동일하고 쿼리는 함수와 동일하다.

명령 - 쿼리 분리 원칙의 요지는 오퍼레이션은 부수효과를 발생시키는 명령이거나 부수효과를 발생시키지 않는 쿼리 중 하나여야 한다는 것이다. 어떤 오퍼레이션도 명령인 동시에 쿼리여서는 안된다. 따라서 명령과 쿼리를 분리하기 위해서는 다음의 두가지 규칙을 준수해야 한다.

- 객체의 상태를 변경하는 명령은 반환값을 가질 수 없다.
- 객체의 정보를 반환하는 쿼리는 상태를 변경할 수 없다.

명령 - 쿼리 분리 원칙을 한 문장으로 표현하면 "질문이 답변을 수정해서는 안된다"는 것이다. 명령은 상태를 변경시킬 수 있지만 상태를 반환해서는 안된다. 쿼리는 객체의 상태를 반환 할 수 있지만 상태를 변경해서는 안된다.

> 부수효과를 발생시키지 않는 것만을 함수로 제한함으로써 소프트웨어에서 말하는 '함수'의 개념이 일반 수학에서의 개념과 상충되지 않게 한다. 객체를 변경하지만 직접적으로 값을 반환하지 않는 명령과 객체에 대한 정보를 반환하지만 변경하지는 않는 쿼리 간의 명확한 구분을 유지할 것이다.

명령-쿼리 분리 원칙은 객체들을 독립적인 기계로 보는 객체지향의 오랜 전통에 기인한다.

버트란드 마이어는 명령-쿼리 분리 원칙을 설명할 때 기계 메타포를 이용한다. 이 관점에서 객체는 블랙박스이며 객체의 인터페이스는 객체의 관찰 가능한 상태를 보기 위한 일련의 디스플레이와 객체의 상태를 변경하기 위해 누를 수 있는 버튼의 집합이다. 이런 스타일의 인터페이스를 사용함으로써 객체의 캡슐화와 다양한 문맥에서의 재사용을 보장할 수 있다. 마틴 파울러는 명령-쿼리 분리 원칙에 따라 작성된 객체의 인터페이스를 명령-쿼리 인터페이스라고 부른다.

반복 일정의 명령과 쿼리 분리하기

도메인의 중요한 두 가지 용어인 "이벤트"와 "반복 일정"에 관해 살펴보자.

"이벤트"는 특정 일자에 실제로 발생하는 사건을 의미한다.

"반복 일정"은 일주일 단위로 돌아오는 특정 시간 간격에 발생하는 사건 전체를 포괄적으로 지칭하는 용어다.

이벤트라는 개념은 Event 클래스로 구현된다. Event 클래스는 이벤트 주제(subject), 시작 일시(from), 소요시간(duration)을 인스턴스 변수로 포함하는 간단한 클래스다.

```
public class Event {
    private String subject;
    private LocalDateTime from;
    private Duration duration;

    public Event(String subject, LocalDateTime from, Duration duration) {
        this.subject = subject;
        this.from = from;
        this.duration = duration;
    }
}
```

"2019년 5월 8일 수요일 10시 30분부터 11시까지 열리는 회의"를 표현하는 Event의 인스턴스는 다음과 같이 생성할 수 있다.

```
Event meeting = new Event("회의",
                          LocalDateTime.of(2019, 5, 8, 10, 30),
                          Duration.ofMinutes(30));
```

"반복 일정"은 RecurringSchedule 클래스로 구현한다. 앞에서 설명한 것처럼 RecurringSchedule은 주 단위로 반복되는 일정을 정의하기 위한 클래스다. 따라서 일정의 주체(subject)와 반복될 요일(dayOfWeek), 시작 시간(from), 기간(duration)을 인스턴스 변수로 포함한다.

```
public class RecurringSchedule {
    private String subject;
    private DayOfWeek dayOfWeek;
    private LocalTime from;
    private Duration duration;

    public RecurringSchedule(String subject, DayOfWeek dayOfWeek,
                             LocalTime from, Duration duration) {
```

```
        this.subject = subject;
        this.dayOfWeek = dayOfWeek;
        this.from = from;
        this.duration = duration;
    }

    public DayOfWeek getDayOfWeek() {
        return dayOfWeek;
    }

    public LocalTime getFrom() {
        return from;
    }

    public Duration getDuration() {
        return duration;
    }
}
```

다음은 RecurringSchedule 클래스를 이용해 "매주 수요일 10시 30분부터 30분 동안 열리는 회의"에 대한 인스턴스를 생성한 코드다.

```
RecurringSchedule schedule = new RecurringSchedule("회의", DayOfWeek.WEDNESDAY,
                                                   LocalTime.of(10, 30),
                                                   Duration.ofMinutes(30));
```

Event 클래스는 현재 이벤트가 RecurringSchedule이 정의한 반복 일정 조건을 만족하는지를 검사하는 isSatisfied 메서드를 제공한다. 이 메서드는 RecurringSchedule의 인스턴스를 인자로 받아 해당 이벤트가 일정 조건을 만족하면 true를, 만족하지 않으면 false를 반환한다.

다음은 isSatisfied 메서드를 사용해 이벤트가 반복 조건을 만족시키는지를 체크하는 코드를 작성한 것이다. 먼저 "매주 수요일 10시 30분 부터 30분 동안 진행되는 회의" 에 대한 반복 일정을 위한 RecurringSchedule 인스턴스를 생성한다. 다음으로 "2019년 5우러 8일 10시 30분 부터 30분 동안 진행되는 회의"를 위한 Event 인스턴스를 생성한다. 5월 8일은 수요일 이므로 반복 일정의 조건을 만족시키기 때문에 isSatisifed 메서드는 true를 반환한다.

```
RecurringSchedule schedule = new RecurringSchedule("회의", DayOfWeek.WEDNESDAY,
                                                   LocalTime.of(10, 30),

Duration.ofMinutes(30));
Event meeting = new Event("회의", LocalDateTime.of(2019, 5, 8, 10, 30),
                          Duration.ofMinutes(30));
assert meeting.isSatisfied(schedule) == true;
```

안타깝게도 이 isSatisfied 메서드 안에 개발팀을 그토록 당혹하게 만들었던 버그가 숨겨져 있다. 다음 코드를 보자.

```
RecurringSchedule schedule = new RecurringSchedule("회의", DayOfWeek.WEDNESDAY,
                                                   LocalTime.of(10, 30),

Duration.ofMinutes(30));
Event meeting = new Event("회의", LocalDateTime.of(2019, 5, 9, 10, 30),
                                  Duration.ofMinutes(30));

assert meeting.isSatisfied(schedule) == false;
assert meeting.isSatisfied(schedule) == true;
```

이 코드는 "매주 수요일 10시 30분 부터 30분 동안 진행되는 회의"에 대한 반복 일정을 표현하는 RecurringSchedule 인스턴스를 생성한다. 다음으로 "2019년 5월 9일 10시 30분부터 30분 동안 진행되는 회의"를 위한 Event 인스턴스인 meeting을 생성한다. 2019년 5월 9일은 목요일 이므로 수요일이라는 반복 일정의 조건을 만족시키지 못한다. 따라서 결과는 false를 반환한다.

흥미로운 부분은 여기부터다. 다시 한 번 isSatisifed 메서드를 호출하면 놀랍게도 true를 반환한다. 이것이 개발팀을 그렇게 괴롭혔던 버그의 정체다. 동일한 Event와 동일한 RecurringSchedule을 이용해 isSatisfied 메서드를 두번 호출했을 때 각 결과가 다른 이유는 무엇일까? 버그의 정체를 파악하기 위해 isSatisfied 메서드를 파헤쳐 보자.

```
public class Event {
    public boolean isSatisfied(RecurringSchedule schedule) {
        if (from.getDayOfWeek() != schedule.getDayOfWeek() ||
            !from.toLocalTime().equals(schedule.getForm()) ||
            !duration.equals(schedule.getDuration())) {

            reschedule(schedule);
            return false;
        }

        return true;
    }
}
```

isSatisfied 메서드는 먼저 인자로 전달된 RecurringSchedule의 요일, 시작 시간, 소요 시간이 현재 Event의 값과 동일한지 판단한다. 이 메서드는 이 값들 중 하나라도 같지 않다면 false를 반환한다. 하지만 false를 반환하기 전에 reschedule 메서드를 호출하고 있다는 사실에 주목하라. 안타깝게도 이 메서드는 Event 객체의 상태를 수정한다.

```
public class Event {
    private void reschedule(RecurringSchedule schedule) {
        from =
LocalDateTime.of(from.toLocalDate().plusDays(daysDistance(schedule)),
                                schedule.getFrom));
        duration = schedule.getDuration();
    }

    private long daysDistance(RecurringSchedule schedule) {
        return schedule.getDayOfWeek().getValue() -
from.getDayOfWeek().getValue();
    }
}
```

reschedule 메서드는 Event의 일정을 인자로 전달된 RecurringSchedule의 조건에 맞게 변경한다. 따라서 reschedule메서드를 호출하는 isSatisfied 메서드는 Event가 RecurringSchedule에 설정된 조건을 만족하지 못할 경우 Event의 상태를 조건을 만족시키도록 변경한 후(여기가 문제다!) false를 반환한다. 예를 들어, 2019년 5월 9일에 일어나는 Event의 isSatisfied 메서드에 매주 수요일마다 반복적으로 발생하는 일정을 가리키는 RecurringSchedule을 전달할 경우 Event의 시작 일자는 2019년 5월 8일로 변경되고 반환 값으로 false가 반환되는 것이다.

버그를 찾기 어려웠던 이유는 isSatisfied가 명령과 쿼리의 두 가지 역할을 동시에 수행하고 있었기 때문이다.

- isSatisfied 메서드는 Event가 RecurringSchedule의 조건에 부합하는지를 판단한 후 부합할 경우 true를 부합하지 않을 경우 false를 반환한다. 따라서 isSatisfied 메서드는 개념적으로 쿼리다.
- isSatisfied 메서드는 Event가 RecurringSchedule의 조건에 부합하지 않을 경우 Event의 상태를 조건에 부합하도록 변경한다. 따라서 isSatisfied는 실제로는 부수효과를 가지는 명령이다.

대부분의 사람들은 isSatisfied 메서드가 부수효과를 가질 것이라고 예상하지 못할 것이다.사실 isSatisfied 메서드가 처음 구현했을 때는 그 안에서 reschedule 메서드를 호출하는 부분이 없었지만 기능을 추가하는 과정에서 조건에 맞지 않을 경우 Event의 상태를 수정해야 한다는 요구사항을 추가했고, 기존에 있던 isSatisfied 메서드에 reschedule 메서드를 호출하는 코드를 추가해 버린 것이다.

명령과 쿼리를 뒤섞으면 실행 결과를 예측하기가 어려워질 수 있다. isSatisfied 메서드처럼 겉으로 보기에는 쿼리처럼 보이지만 내부적으로 부수효과를 가지는 메서드는 이해하기 어렵고, 잘못 사용하기 쉬우며, 버그를 양산하는 경향이 있다.

가장 깔끔한 해결책은 명령과 쿼리를 명확하게 분리하는 것이다.

```
public class Event {
    public boolean isSatisfied(RecurringSchedule schedule) {
       if (from.getDayOfWeek() != schedule.getDayOfWeek() ||
            !from.toLocalTime().equals(schedule.getForm()) ||
            !duration.equals(schedule.getDuration())) {

            reschedule(schedule);
            return false;
        }

        return true;
    }
```

```
    public void reschedule(RecurringSchedule schedule) {
        from =
LocalDateTime.of(from.toLocalDate().plusDays(daysDistance(schedule)),
                                 schedule.getFrom));
        duration = schedule.getDuration();
    }
}
```

수정 후의 isSatisfied 메서드는 부수효과를 가지지 않기 때문에 순수한 쿼리가 됐다. 이제 Event의 인터페이스를 살펴보면 isSatisfied 메서드는 반환 값을 돌려주고 reschedule 메서드는 반환 값을 돌려주지 않는다. Event는 현재 명령과 쿼리를 분리한 상태이므로 인터페이스를 훑어보는 것만으로도 isSatisfied 메서드가 쿼리이고, reschedule 메서드가 명령이라는 사실을 한눈에 알 수 있다.

```
public class Event {
    public boolean isSatisfied(RecurringSchedule schedule) {...}
    public void reschedule(RecurringSchedule schedule) {...}
}
```

- 반환 값을 돌려주는 메서드는 쿼리이므로 부수 효과에 대한 부담이 없다. 따라서 몇 번을 호출하더라도 다른부분에 영향을 미치지 않는다. 반면 반환 값을 가지지 않는 메서드는 모두 명령이므로 해당 메서드를 호출할 때는 부수효과에 주의해야 한다. 어떤 메서드가 부수효과를 가지는지를 확인하기 위해 코드를 일일이 분석하는 것보다는 메서드가 반환 값을 가지는지 여부만 확인하는 것이 훨씬 간단하지 않은가?
- 명령과 쿼리를 분리하면서 reschedule 메서드의 가시성이 private에서 public으로 변경됐다는 점을 눈여겨보기 바란다. 이것은 원래 isSatisfied 메서드 안에서 수행했던 명령을 클라이언트가 직접 실행 할 수 있게 하기 위해서다.

```
if (!event.isSatisfied(schedule)) {
    event.reschedule(schedule);
}
```

수정 전보다 Event의 상태를 변경하기 위한 인터페이스가 더 복잡해진 것 처럼 보이지만 이 경우에는 명령과 쿼리를 분리함으로써 얻는 이점이 더 크다. 퍼블릭 인터페이스를 설계할 때 부수효과를 가지는 대신 값을 반환하지 않는 명령과 부수효과를 가지지 않는 대신 값을 반환하는 쿼리를 분리하길 바란다.

그 결과 코드는 예측 가능하고 이해하기 쉬우며 디버깅이 용이한 동시에 유지보수가 수월해질 것이다.

명령 - 쿼리 분리와 참조 투명성

지금까지 살펴본 것처럼 명령과 쿼리를 엄격하게 분류하면 객체의 부수효과를 제어하기가 수월해진다.

쿼리는 객체의 상태를 변경하지 않기 때문에 몇 번이고 반복적으로 호출하더라도 상관이 없다. 명령이 개입하지 않는 한 쿼리의 값은 변경되지 않기 때문에 몇 번이고 반복적으로 호출하더라도 상관이 없다. 명령이 개입하지 않는 한 쿼리의 값은 변경되지 않기 때문에 쿼리의 결과를 예측하기 쉬워진다. 또한 쿼리들의 순서를 자유롭게 변경할 수도 있다.

명령과 쿼리를 분리함으로써 명령형 언어의 틀 안에서 **참조 투명성**의 장점을 제한적이나마 누릴 수 있게 된다. 참조 투명성이라는 특성을 잘 활용하면 버그가 적고, 디버깅이 용이하며, 쿼리의 순서에 따라 실행 결과가 변하지 않는 코드를 작성할 수 있다.

컴퓨터의 세계와 수학의 세계를 나누는 가장 큰 특징은 **부수효과**의 존재 유무다. 프로그램에서 부수효과를 발생시키는 두 가지 대표적인 문법은 대입문과 (원래는 프로시저라고 불려야 올바른) 함수다. 수학의 경우 x의 값을 초기화한 후에는 값을 변경하는 것이 불가능하지만 프로그램에서는 대입문을 이용해 다른값으로 변경하는 것이 가능하다. 함수는 내부에 부수효과를 포함할 경우 동일한 인자를 전달하더라도 부수효과에 의해 그 결괏값이 매번 달라질 수 있다.

- 부수효과를 이야기할 때 빠질 수 없는 것이 바로 **참조 투명성**이다. 참조 투명성이란 "어떤 표현식 e가 있을 때 e의 값으로 e의 값으로 교체하더라도 결과가 달라지지 않는 특성"을 의미한다.
- EX) f(1) = 3, f(1) + f(1) = 6, 3 + 3 = 6

f(1) 의 값이 3이면 동일한 입력에 대해 항상 동일한 값을 출력하는 함수의 특성은 아무런 걱정없이 모든 f(1)을 3으로 대체하는 것을 허용한다.

f(1)의 값을 항상 3이라고 말할 수 있는 이유는 f(1)의 값이 변하지 않기 때문이다. 이처럼 어떤 값이 변하지 않는 성질을 **불변성**이라고 부른다. 어떤 값이 불변한다는 말은 부수효과가 발생하지 않는다는 말과 동일하다.

불변성은 부수효과의 발생을 방지하고 참조 투명성을 만족시킨다.

참조 투명성의 또 다른장점은 식의 순서를 변경하더라도 결과가 달라지지 않는다는 것이다.

참조 투명성을 만족하는 식은 우리에게 두 가지 장점을 제공한다.

1. 모든 함수를 이미 알고 있는 하나의 결괏값으로 대체할 수 있기 때문에 식을 쉽게 계산할 수 있다.
2. 모든 곳에서 함수의 결괏값이 동일하기 때문에 식의 순서를 변경하더라도 각 식의 결과는 달라지지 않는다.

객체지향 패러다임이 객체의 상태 변경이라는 부수효과를 기반으로 하기 때문에 참조 투명성은 예외에 가깝다.

명령 - 쿼리 분석 원칙을 사용하면 이 균열을 조금이나마 줄일 수 있다. 명령 - 쿼리 분리 원칙은 부수효과를 가지는 명령으로부터 부수효과를 가지지 않는 쿼리를 명백하게 분리함으로써 제한적이나마 참조 투명성의 혜택을 누릴 수 있게 된다.

> 부수효과를 기반으로 하는 프로그래밍 방식을 명령형 프로그래밍 이라고 부른다. 명령형 프로그래밍은 상태를 변경시키는 연산들을 적절한 순서대로 나열함으로써 프로그램을 작성한다. 대부분의 객체지향 프로그래밍 언어들은 메시지에 의한 객체의 상태 변경에 집중하기 때문에 명령형 프로그래밍 언어로 분류된다.

> 최근 들어 주목받고 있는 함수형 프로그래밍은 부수효과가 존재하지 않는 수학적인 함수에 기반한다. 따라서 함수형 프로그래밍에서는 참조 투명성의 장점을 극대화할 수 있으며 명령형 프로그래밍에 비해 프로그램의 실행결과를 이해하고 예측하기가 더 쉽다. 또한 하드웨어의 발달로 병렬 처리가 중요해진 최근엔 함수형 프로그래밍의 인가가 상승하고 있으며 다양한 객체지향 언어들이 함수형 프로그래밍 패러다임을 접목시키고 있는 추세다.

책임에 초점을 맞춰라

디미터 법칙을 준수하고 묻지말고 시켜라 스타일을 따르면서도 의도를 드러내는 인터페이스를 설계하는 아주 쉬운 방법이 있다. 메시지를 먼저 선택하고 그 후에 메시지를 처리할 객체를 선택하는 것이다.

명령과 쿼리를 분리하고 계약에 의한 설계 개념을 통해 객체의 협력 방식을 명시적으로 드러낼 수 있는 방법은 객체의 구현 이전에 객체 사이의 협력에 초점을 맞추고 협력 방식을 단순하고, 유연하게 만드는 것이다.

이 모든 방식의 중심에는 객체가 수행할 책임이 위치한다.

메시지를 먼저 선택하는 방식이 디미터 법칙, 묻지 말고 시켜라 스타일, 의도를 드러내는 인터페이스,

명령 - 쿼리 분리 원칙에 미치는 긍정적인 영향을 살펴보면 다음과 같다

- 디미터 법칙 : 협력이라는 컨텍스트 안에서 객체보다 메시지를 먼저 결정하면 두 객체 사이의 구조적인 결합도를 낮출 수 있다.
- 묻지 말고 시켜라 : 메시지를 먼저 선택하면 묻지 말고 시켜라 스타일에 따라 협력을 구조화하게 된다. 클라이언트 관점에서 메시지를 선택하기 때문에 필요한 정보를 물을 필요 없이 원하는 것을 표현한 메시지를 전송하면 된다.
- 의도를 드러내는 인터페이스 : 메시지를 먼저 선택한다는 것은 메시지를 전송하는 클라이언트 관점에서 메시지의 이름을 정한다는 것이다. 당연히 그 이름에는 클라이언트가 무엇을 원하는지, 그 의도가 분명히 드러날 수 밖에 없다.
- 명령 - 쿼리 분리 원칙 : 메시지를 먼저 선택한다는 것은 협력이라는 문맥 안에서 객체의 인터페이스에 관해 고민한다는 것을 의미한다. 객체가 단순히 어떤 일을 해야 하는지뿐만 아니라 협력 속에서 객체의 상태를 예측하고 이해하기 쉽게 만들기 위한 방법에 관해 고민하게 된다. 따라서 예측 가능한 협력을 만들기 위해 명령과 쿼리를 분리하게 될 것이다.

훌륭한 메시지를 얻기 위한 출발점은 책임 주도 설계 원칙을 따르는 것이다. 책임 주도 설계에서는 객체가 메시지를 선택하는 것이 아니라 메시지가 객체를 선택하기 때문에 협력에 적합한 메시지를 결정 할 수 있는 확률이 높아진다. 우리에게 중요한 것은 협력에 적합한 객체가 아니라 협력에 적합한 메시지다.

책임 주도 설계 방법에 따라 메시지가 객체를 결정하게 하라. 그러면 여러분의 설계가 아름답고 깔끔해지며 심지어 우아해 진다는 사실을 실감하게 될 것이다.