

부록 A. 계약에 의한 설계

계약에 의한 설계

명령과 쿼리를 분리*해도 명령 때문에 발생하는 부수효과를 명확히 표현하긴 어렵다

구현이 단순하다면 부수효과를 쉽게 파악할 수 있지만 구현이 복잡하고 부수효과를 가진 메서드들을 연달아 호출한다면 실행 결과를 예측하긴 너무 어렵다

계약에 의한 설계를 이용하면 협력에 필요한 제약이나 부수효과를 명확히 표현할 수 있고 문서화도 할 수 있다.

이번 장에선 계약에 의한 설계를 사용하는 이유와 장점에 대해 이해해보자!

* 명령 - 쿼리 분리

- 어떤 오퍼레이션도 명령인 동시에 쿼리어선 안 된다. 둘을 명확하게 분리하라. 명령과 쿼리가 섞인 메서드는 실행 결과를 예측하기가 어렵다.
- 명령 : 객체의 상태를 수정하는 오퍼레이션이다. 반환값을 가질 수 없다.
- 쿼리 : 객체와 관련된 정보를 반환하는 오퍼레이션이다. 상태를 변경할 수 없다.

1. 협력과 계약

부수효과를 명시적으로 표현 가능

6장 일정 관리 프로그램에서 `isSatisfied()`로 이벤트가 현재 일정 규칙과 맞는지 확인했었다.

이때 계약에 의한 설계 라이브러리인 Code Contracts를 사용하면 `IsSatisfied` 실행 결과가 `true`일 때만 `Reschedule` 메서드를 호출할 수 있단 걸 명확히 알 수 있다.

```
class Event {  
    public bool IsSatisfied(RecurringSchedule schedule) {...}  
  
    public void Reschedule(RecurringSchedule schedule) {  
        Contract.Requires(IsSatisfied(schedule)); //일반 로직과 명확히 구분된다(!)  
        ...  
    }  
}
```

1. 협력과 계약

계약

계약의 특성은 다음과 같다.

- 각 계약 당사자는 계약으로부터 이익을 기대하고, 이익을 얻기 위해 의무를 이행한다.
- 각 계약 당사자의 이익과 의무는 계약서에 문서화된다.

💡 이때 주목할 부분은

- 한쪽의 의무가 반대쪽의 권리가 된다.
- 계약을 이행하는 구체적인 방식에 대해선 간섭하지 않는다.

이러한 특성이 계약에 의한 설계에도 적용된다.

2. 계약에 의한 설계

계약에 의한 설계 개념은 “인터페이스에 대해 프로그래밍하라”는 원칙을 확장한 것이다.

계약에 의한 설계를 이용하면 오퍼레이션의 시그니처를 이용해 인터페이스의 일부로 만들어서 인터페이스의 사용법을 이해할 수 있다.

가시성	메서드 이름			
<code>public</code>	<code>Reservation</code>	<code>reserve</code>	<code>(Customer customer, int audienceCount)</code>	<code>{...}</code>
	반환타입		파라미터 타입, 이름	

의도를 드러내는 인터페이스를 통해 오퍼레이션의 시그니처만으로도 어느 정도 제약 조건을 명시할 수 있다.

그러나 계약은 여기서 더 나아간다!

예약을 만들 때 예약 고객이 null이어선 안 되며, 관객 수는 1보다 크거나 같아야 한다. 이 파라미터를 받아 만들어진 예약도 null이어선 안 된다.

2. 계약에 의한 설계

서버는 자신이 처리할 수 있는 범위의 값들을 클라이언트가 전달할 것이라고 기대하며, 클라이언트는 자신이 원하는 값을 서버가 반환할 거라고 기대한다. 클라이언트는 메세지 전송 전 / 후 서버의 상태가 정상일 것이라고 기대한다.

이 세가지가 바로 계약에 의한 설계를 구성하는 세 가지 요소에 대응된다.

- **사전조건**: 메서드가 호출되기 위해 만족되어야 하는 조건. 이걸 클라이언트의 의무다.
- **사후조건**: 메서드가 실행된 후에 클라이언트에게 보장해야 하는 조건. 이걸 서버의 의무다.
- **불변식**: 항상 참이라고 보장되는 서버의 조건. 메서드가 실행되는 도중에는 불변식을 만족시키지 못할 수도 있지만 메서드를 실행하기 전 / 종료된 후에 불변식은 항상 참이어야 한다.

사전조건, 사후조건, 불변식을 기술할 땐 실행 절차 없이 **상태 변경만을 명시**한다. → 코드를 이해하고 분석하기 쉬워짐 👍

2. 계약에 의한 설계

사전조건

일반적으로 메서드에 전달된 인자의 적합성을 체크할 때 사용한다.

```
public Reservation Reserve(Customer customer, int audienceCount) {  
    Contract.Requires(customer != null);  
    Contract.Requires(audienceCount >= 1);  
    return new Reservation(...)  
}
```

사전 조건을 만족시킬 책임은 `Reserve` 메서드를 호출하는 클라이언트에 있단 사실을 기억하라!

클라이언트가 사전 조건을 만족시키지 못하면 `Reserve` 메서드는 최대한 빨리 실패해서 클라이언트에게 값을 잘못 전달했음을 알린다

`Contract.Requires` 메서드는 클라이언트가 계약의 조건을 만족시키지 못했을 때 `ContractException` 예외를 발생시킨다

2. 계약에 의한 설계

사전조건


일반적으로 메서드에 전달된 인자의 적합성을 체크할 때 사용한다.

```
public Reservation Reserve(Customer customer, int audienceCount) {  
    Contract.Requires(customer != null);  
    Contract.Requires(audienceCount >= 1);  
    return new Reservation(...)  
}
```

사전 조건을 만족시킬 책임은 **Reserve 메서드를 호출하는 클라이언트**에 있단 사실을 기억하라!

클라이언트가 사전 조건을 만족시키지 못하면 Reserve 메서드는 **최대한 빨리 실패**해서 클라이언트에게 값을 잘못 전달했음을 알린다

Contract.Requires 메서드는 클라이언트가 계약의 조건을 만족시키지 못했을 때 **ContractException 예외**를 발생시킨다



문제가 발생한 그 위치에서 프로그램이 실패하도록 만들어라.
문제의 원인을 가장 빠르게 파악할 수 있는 방법이다!

2. 계약에 의한 설계

사전조건

일반적으로 메서드에 전달된 인자의 적합성을 체크할 때 사용한다.

```
public Reservation Reserve(Customer customer, int audienceCount) {  
    Contract.Requires(customer != null);  
    Contract.Requires(audienceCount >= 1);  
    return new Reservation(...)  
}
```

사전 조건을 만족시킬 책임은 **Reserve 메서드를 호출하는 클라이언트**에 있단 사실을 기억하라!

클라이언트가 사전 조건을 만족시키지 못하면 Reserve 메서드는 **최대한 빨리 실패**해서 클라이언트에게 값을 잘못 전달했음을 알린다

Contract.Requires 메서드는 클라이언트가 계약의 조건을 만족시키지 못했을 때 **ContractException 예외**를 발생시킨다

문제가 발생한 그 위치에서 프로그램이 실패하도록 만들어라.
문제의 원인을 가장 빠르게 파악할 수 있는 방법이다!

```
screening.Reserve(null, 2); //ContractException 예외 발생!
```

2. 계약에 의한 설계

사후조건

일반적으로 (1) 인스턴스 변수의 상태가 올바른지 (2) 메서드에 전달된 파라미터의 값이 올바르게 변경됐는지 (3) 반환값이 올바른지 서술하기 위해 사용된다.

```
public Reservation Reserve(Customer customer, int audienceCount) {  
    Contract.Requires(customer != null);  
    Contract.Requires(audienceCount >= 1);  
    Contract.Ensures(Contract.Result<Reservation>() != null);  
    return new Reservation(...)  
}
```

사후 조건 정의에 `Contract.Ensures` 메서드를 제공한다.

`Contract.Result<T>` 메서드를 통해 `Reserve` 메서드의 실행 결과에 접근할 수 있다.

2. 계약에 의한 설계

사후조건

일반적으로 (1) 인스턴스 변수의 상태가 올바른지 (2) 메서드에 전달된 파라미터의 값이 올바르게 변경됐는지 (3) 반환값이 올바른지 서술하기 위해 사용된다.

```
public String Middle(string text) {  
    Contract.Requires(text != null && text.Length >= 2);  
    // (as-is) text 값이 메서드 실행 중에 변경되므로 사후조건 체크 어려움  
    // Contract.Ensures(Contract.Result<string>().Length < text.Length);  
    Contract.Ensures(Contract.Result<string>().Length < Contract.OldValue<string>(text).Length); //사후 조건 정상  
    체크 👍  
    text = text.Substring(1, text.Length - 2);  
    return text.Trim();  
}
```

Contract.OldValue<T>를 이용하면 메서드를 실행 전의 상태에 접근 할 수 있다.

2. 계약에 의한 설계

사후조건

일반적으로 (1) 인스턴스 변수의 상태가 올바른지 (2) 메서드에 전달된 파라미터의 값이 올바르게 변경됐는지 (3) 반환값이 올바른지 서술하기 위해 사용된다.

(1) 인스턴스 변수의 상태가 올바른지

```
java

public class Counter {
    private int count = 0;

    public void increment() {
        int oldCount = count; // 이전 상태 저장
        count++;

        // 사후조건: count가 정확히 1 증가했는지 확인
        assert count == oldCount + 1 : "count가 올바르게 증가하지 않음";
        assert count >= 0 : "count는 항상 0 이상이어야 함";
    }
}
```

(2) 메서드 파라미터의 값이 올바르게 변경됐는지

```
java

public void sortArray(int[] array) {
    int[] originalArray = array.clone(); // 원본 배열 복사

    Arrays.sort(array); // 배열 정렬

    // 사후조건: 파라미터로 전달된 배열이 올바르게 변경되었는지
    assert array.length == originalArray.length : "배열 크기가 변경됨";
    assert Arrays.equals(Arrays.stream(array).sorted().toArray(),
                          Arrays.stream(originalArray).sorted().toArray())
        : "배열 요소가 손실됨";
    assert isSorted(array) : "배열이 정렬되지 않음";
}
```

2. 계약에 의한 설계

불변식

불변식은 객체가 생성된 후부터 소멸될 때까지 항상 참이어야 하는 조건이다. 객체의 "생존 규칙"으로 생각해도 된다!

[🏦 은행 계좌 예시]

```
public class BankAccount {  
    private double balance;  
    private String accountNumber;  
  
    // 불변식:  
    // 1. balance >= 0 (잔액은 항상 0 이상)  
    // 2. accountNumber != null (계좌번호는 항상 존재)  
    // 3. accountNumber.length() == 10 (계좌번호는 항상 10자리)  
}
```

2. 계약에 의한 설계

불변식

불변식은 객체가 생성된 후부터 소멸될 때까지 항상 참이어야 하는 조건이다. 객체의 "생존 규칙"으로 생각해도 된다!

[📊 성적 관리 시스템 예시]

```
public class StudentGrade {  
    private int score;  
    private String grade;  
  
    // 불변식:  
    // 1.  $0 \leq \text{score} \leq 100$  (점수는 0~100 사이)  
    // 2. score와 grade가 일치해야 함  
    //    (90~100: A, 80~89: B, 70~79: C, ...)  
}
```

불변식은 **생성자 실행 후**, **메서드 실행 전**, **메서드 실행 후에** 호출돼야 한다!!

2. 계약에 의한 설계

불변식

```
public class Screening {  
    private Movie movie;  
    private int sequence;  
    private DateTime whenScreened;  
  
    [ContractInvariantMethod]  
    private void Invariant() {  
        Contract.Invariant(movie != null);  
        Contract.Invariant(sequence >= 1);  
        Contract.Invariant(whenScreened > DateTime.Now);  
    }  
}
```

메서드에 `ContractInvariantMethod`를 지정하면 매번 생성자 실행 후 / 메서드 실행 전 / 후에 추가할 필요없이 자동 추가된다

`Contract.Invariant` 메서드로 불변식을 정의할 수 있다.

3. 계약에 의한 설계와 서브타이핑

계약에 의한 설계의 핵심은 **클라이언트 - 서버 간 준수해야 하는 규약**을 정의하는 것이다.

리스코프 치환 원칙은 슈퍼타입의 인스턴스와 협력하는 클라이언트의 관점에서 서브타입의 인스턴스가 슈퍼타입을 대체하더라도 협력에 지장이 없어야 한다는 것이다.

즉, 서브타입이 **리스코프 치환 원칙**을 만족시키기 위해선 **클라이언트와 슈퍼타입 간에 체결된 계약을 준수해야 한다**.

3. 계약에 의한 설계와 서브타이핑

리스코프 치환 원칙의 규칙은 두 종류로 세분화할 수 있다. 규칙을 어긴다면 계약 위반이며 즉 리스코프 치환 원칙 위반이다.

1. 계약 규칙 : 슈퍼타입 - 서브타입 간 사전조건, 사후조건, 불변식에 대한 제약에 관한 규칙이다. 리스코프 치환 원칙과 관련됨.

- 서브타입에 더 강력한 사전조건 정의 불가
- 서브타입에 더 완화된 사후조건 정의 불가
- 슈퍼타입의 불변식은 서브타입에서도 반드시 유지되어야 함

2. 가변성 규칙 : 교체 가능한 타입과 관련된 규칙이다. 파라미터와 리턴 타입의 변형과 관련된 규칙이다.

- 서브타입의 메서드 파라미터는 반공변성을 가져야 함
- 서브타입의 리턴타입은 공변성을 가져야 함
- 서브타입은 슈퍼타입이 발생시키는 예외와 다른 타입의 예외를 발생시켜선 안 됨

3. 계약에 의한 설계와 서브타이핑

계약 규칙 3. 슈퍼타입의 불변식은 서브타입에서도 반드시 유지되어야 한다.

```
public abstract class AdditionalRatePolicy implements RatePolicy {

    protected RatePolicy next;

    public AdditionalRatePolicy(RatePolicy next) {
        this.next = next;
        assert next != null; // 불변식. 다음 요금제를 가리키는 next는 null이면 안 된다
    }

    @Override
    public Money calculateFee(List<Call> calls) {
        assert next != null; // 불변식
        assert calls != null; // 사전조건
        . . .
        assert result.isGreaterThanOrEqualTo (Money.ZERO); // 사후조건
        assert next != null; // 불변식
        return result;
    }
}
```

3. 계약에 의한 설계와 서브타이핑

계약 규칙 3. 슈퍼타입의 불변식은 서브타입에서도 반드시 유지되어야 한다.

```
public abstract class AdditionalRatePolicy implements RatePolicy {  
  
    protected RatePolicy next;  
  
    public AdditionalRatePolicy(RatePolicy next) {  
        this.next = next;  
        assert next != null; // 불변식. 다음 요금제를 가리키는 next는 null이  
    }  
  
    @Override  
    public Money calculateFee(List<Call> calls) {  
        assert next != null; // 불변식  
        assert calls != null; // 사전조건  
        . . .  
        assert result.isGreaterThanOrEqual (Money.ZERO); // 사후조건  
        assert next != null; // 불변식  
        return result;  
    }  
}
```



취약점 존재!!

3. 계약에 의한 설계와 서브타이핑

계약 규칙 3. 슈퍼타입의 불변식은 서브타입에서도 반드시 유지되어야 한다.

```
public abstract class AdditionalRatePolicy implements RatePolicy {  
  
    protected RatePolicy next; // Additional의 자식 클래스는 부모 클래스 몰래 next 값을 수정할 수 있다(;)   
  
    public AdditionalRatePolicy(RatePolicy next) {  
        this.next = next;  
        assert next != null; // 불변식. 다음 요금제를 가리키는 next는 null이면 안 된다  
    }  
  
    @Override  
    public Money calculateFee(List<Call> calls) {  
        assert next != null; // 불변식  
        assert calls != null; // 사전조건  
        . . .  
        assert result.isGreaterThanOrEqualTo (Money.ZERO); // 사후조건  
        assert next != null; // 불변식  
        return result;  
    }  
}
```

3. 계약에 의한 설계와 서브타이핑

계약 규칙 3. 슈퍼타입의 불변식은 서브타입에서도 반드시 유지되어야 한다.

```
public class RateDiscountablePolicy extends AdditionalRatePolicy {  
    public void changeNext(RatePolicy next) {  
        this.next = next; // protected 변수이므로 RateDP에서 next 값 변경이 가능 → 불변성 위반  
    }  
}
```

이 예를 통해 계약의 관점에서 캡슐화의 중요성을 알 수 있다.

자식 클래스가 계약을 위반할 수 있는 코드를 작성하는 걸 막는 유일한 방법은 인스턴스 변수의 가시성을 protected -> private으로 만드는 것뿐!

protected 인스턴스 변수를 가진 부모 클래스의 불변성은 자식 클래스에 의해 언제라도 무너질 수 있다.

따라서 모든 인스턴스 변수의 가시성은 private으로 제한해야 한다. ★★★

3. 계약에 의한 설계와 서브타이핑

계약 규칙 3. 슈퍼타입의 불변식은 서브타입에서도 반드시 유지되어야 한다.

만약 자식 클래스에서 인스턴스 변수의 상태를 변경하고 싶다면?

부모 클래스에 `protected` 메서드를 제공하고 이 메서드를 통해 불변식을 체크하게 해야 한다.

```
public abstract class AdditionalRatePolicy implements RatePolicy {  
  
    private RatePolicy next;  
  
    public AdditionalRatePolicy(RatePolicy next) {  
        changeNext(next);  
    }  
  
    protected void changeNext(RatePolicy next) { // ? 자식 클래스에서 @Override한다면 next 값을 수정할 수 있는 거 아닌가  
        this.next = next;  
        assert next != null; // 불변식  
    }  
}
```

3. 계약에 의한 설계와 서브타이핑

가변성 규칙 1. 서브타입은 슈퍼타입이 발생시키는 예외와 다른 타입의 예외를 발생시켜선 안 된다.

```
public class EmptyCallException extends RuntimeException { . . . }

public interface RatePolicy {
    Money calculateFee(List<Call> calls) throws EmptyCallException;
}
```

```
public abstract class BasicRatePolicy implements RatePolicy {
    @Override
    public Money calculateFee(List<Call> calls) {
        if (calls == null || calls.isEmpty()) { throw new EmptyCallException(); } // 빈 리스트를 전달받으면 EmptyCallException
        예외
    }
}
```

```
public void calculate(RatePolicy policy, List<Call> calls) {
    try {
        return policy.calculateFee(calls);
    } catch (EmptyCallException ex) {
        return Money.ZERO; // RatePolicy와 협력하는 메서드가 있다. 이 메서드는 EmptyCallException 예외가 던져지면 캐치해서 0원을
        반환한다
    }
}
```

3. 계약에 의한 설계와 서브타이핑

가변성 규칙 1. 서브타입은 슈퍼타입이 발생시키는 예외와 다른 타입의 예외를 발생시켜선 안 된다.

```
public abstract class AdditionalRatePolicy implements RatePolicy {  
    @Override  
    public Money calculateFee(List<Call> calls ) {  
        if (calls == null || calls.isEmpty()) {  
            throw new NoSuchElementException(); // 만약 EmptyCallException이 아닌 NoSuchElementException 을 던진다면?  
        }  
    }  
}
```


3. 계약에 의한 설계와 서브타이핑

가변성 규칙 1. 서브타입은 슈퍼타입이 발생시키는 예외와 다른 타입의 예외를 발생시켜선 안 된다.

```
public abstract class AdditionalRatePolicy implements RatePolicy (  
    @Override  
    public Money calculateFee(List<Call> calls ) {  
        if (calls == null || calls.isEmpty()) {  
            throw new NoSuchElementException(); // 만약 EmptyCallException이 아닌 NoSuchElementException 을 던진다면?  
        }  
    }
```

```
public void calculate(RatePolicy policy, List<Call> calls) {  
    try {  
        return policy.calculatereee(calls);  
    } catch (EmptyCallException ex) { // catch 문에서 잡히지 않음 (NoSuchElementException 클래스가 EmptyCallException 클래스의 자식 클래스가 아니라고 가정)  
        return Money. ZERO;  
    }  
}
```

결과적으로 클라이언트 입장에서 협력의 결과가 예상을 벗어났기 때문에 AdditionalRatePolicy 는 RatePolicy 를 대체할 수 없다.

3. 계약에 의한 설계와 서브타이핑

가변성 규칙 1. 서브타입은 슈퍼타입이 발생시키는 예외와 다른 타입의 예외를 발생시켜선 안 된다.

```
public class Bird {  
    public void fly () { . . . } // 예외 없음  
}  
  
public class Penguin extends Bird {  
    @Override  
    public void fly() {  
        throw new UnsupportedOperationException() // fly() 호출 시 예외 발생  
    }  
}
```

클라이언트는 Bird의 인스턴스에게 fly 메시지를 전송했을 때 UnsupportedOperationException 예외가 던져지리라고 기대하지 않았을 것이다.

따라서 클라이언트의 관점에서 Penguin 은 Bird 를 대체할 수 없다.

3. 계약에 의한 설계와 서브타이핑

가변성 규칙 2. 서브타입의 리턴 타입은 공변성을 가져야 한다.


S가 T의 서브타입이라고 했을 때 프로그램의 어떤 위치에서 두 타입 사이의 치환 가능성을 다음과 같이 나눌 수 있다.

- **공변성**(함께 변한다) : S 와 T 사이의 서브타입 관계가 그대로 유지된다. 이 경우 해당 위치에서 서브타입인 S 가 슈퍼타입인 T 대신 사용될 수 있다. 우리가 흔히 이야기하는 리스코프 치환 원칙은 공변성과 관련된 원칙이다.
- **반공변성**(반대로 변한다) : S 와 T 사이의 서브타입 관계가 역전된다. 이 경우 해당 위치에서 슈퍼타입인 T 가 서브타입인 S 대신 사용될 수 있다.
- **무공변성** : S 와 T 사이에는 아무런 관계도 존재하지 않는다. 따라서 S 대신 T 를 사용하거나 T 대신 S 를 사용할 수 없다.

3. 계약에 의한 설계와 서브타이핑

가변성 규칙 2. 서브타입의 리턴 타입은 공변성을 가져야 한다.

서브타입의 리턴 타입은 공변성을 가져야 한다는 것은 자식 클래스가 메서드를 오버라이드할 때 리턴 타입을 더 구체적인 타입으로 바꿀 수 있다는 뜻!

 음식점 체인

일반식당 → "음식 주세요" → 기본 음식

한식당 (일반식당 상속) → "음식 주세요" → 한식 (더 구체적!)

김치찌개전문점 (한식당 상속) → "음식 주세요" → 김치찌개 (더더욱 구체적!)



핸드폰 매장

전자제품점 → "전자제품 주세요" → 기본 전자제품

휴대폰매장 → "전자제품 주세요" → 휴대폰 (더 좋음!)

아이폰전문점 → "전자제품 주세요" → 아이폰 (최고!)

3. 계약에 의한 설계와 서브타이핑

가변성 규칙 2. 서브타입의 리턴 타입은 공변성을 가져야 한다.

```
class Fruit {  
    public String getName() {  
        return "Fruit";  
    }  
}
```

```
class Apple extends Fruit {  
    @Override  
    public String getName() {  
        return "Apple";  
    }  
}
```

```
class FruitStore {  
    public Fruit getItem() {  
        return new Fruit();  
    }  
}
```

```
class AppleStore extends FruitStore {  
    // 리턴 타입이 부모보다 더 구체적인 Apple!  
    @Override  
    public Apple getItem() {  
        return new Apple();  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        FruitStore store = new AppleStore(); // 업캐스팅  
        Fruit item = store.getItem();        // Apple을 반환하지만 타입은 Fruit → 클라이언트 입장에서 Apple도 Fruit에 속하니까 OK  
        System.out.println(item.getName());  // 결과: Apple  
    }  
}
```

3. 계약에 의한 설계와 서브타이핑

가변성 규칙 3. 서브타입의 메서드 파라미터는 반공변성을 가져야 한다.

서브타입의 메서드 파라미터는 반공변성을 가져야 한다는 것은 자식 클래스가 메서드를 오버라이드할 때 **파라미터를 더 일반화할 수 있다**는 것이다.

```
class Fruit {}
class Apple extends Fruit {}

class FruitHandler {
    public void handle(Fruit fruit) {
        System.out.println("Handling fruit");
    }
}

class AppleHandler extends FruitHandler {

    // 아래처럼 하면 안 됨!
    // public void handle(Apple apple) { ... } // ❌ 파라미터가 부모 클래스의 파라미터보다 더 구체적이다

    // 이렇게 해야 됨 (같거나 더 일반적)
    @Override
    public void handle(Fruit fruit) {
        System.out.println("Handling fruit in AppleHandler");
    }
}
```

클라이언트 입장에서 FruitHandler 타입으로 AppleHandler를 써도 문제 없어야 한다.

```
FruitHandler handler = new AppleHandler();
handler.handle(new Fruit()); // 이게 되어야 함
```

근데 AppleHandler가 handle(Apple)만 받는다면 클라이언트는 Fruit를 받길 기대하는데 받을 수 없게 됨
→ 치환 불가능

4. 정리

계약에 의한 설계는 클래스의 부수효과를 명시적으로 문서화하며 명확하게 커뮤니케이션하는 것을 돕는다. 실행 가능한 검증 도구로써 사용할 수도 있다.

서브타입이 슈퍼타입을 치환할 수 있다 = 계약에 의한 설계에서 정의한 계약 규칙과 가변성 규칙을 준수한다.

진정한 서브타이핑 관계를 만들고 싶다면 사전조건 강화, 사후조건 완화는 금지되며 슈퍼타입의 불변식을 유지하기 위해 항상 노력하라.
또한 서브타입에서 슈퍼타입에서 정의하지 않은 예외를 던져선 안 된다.