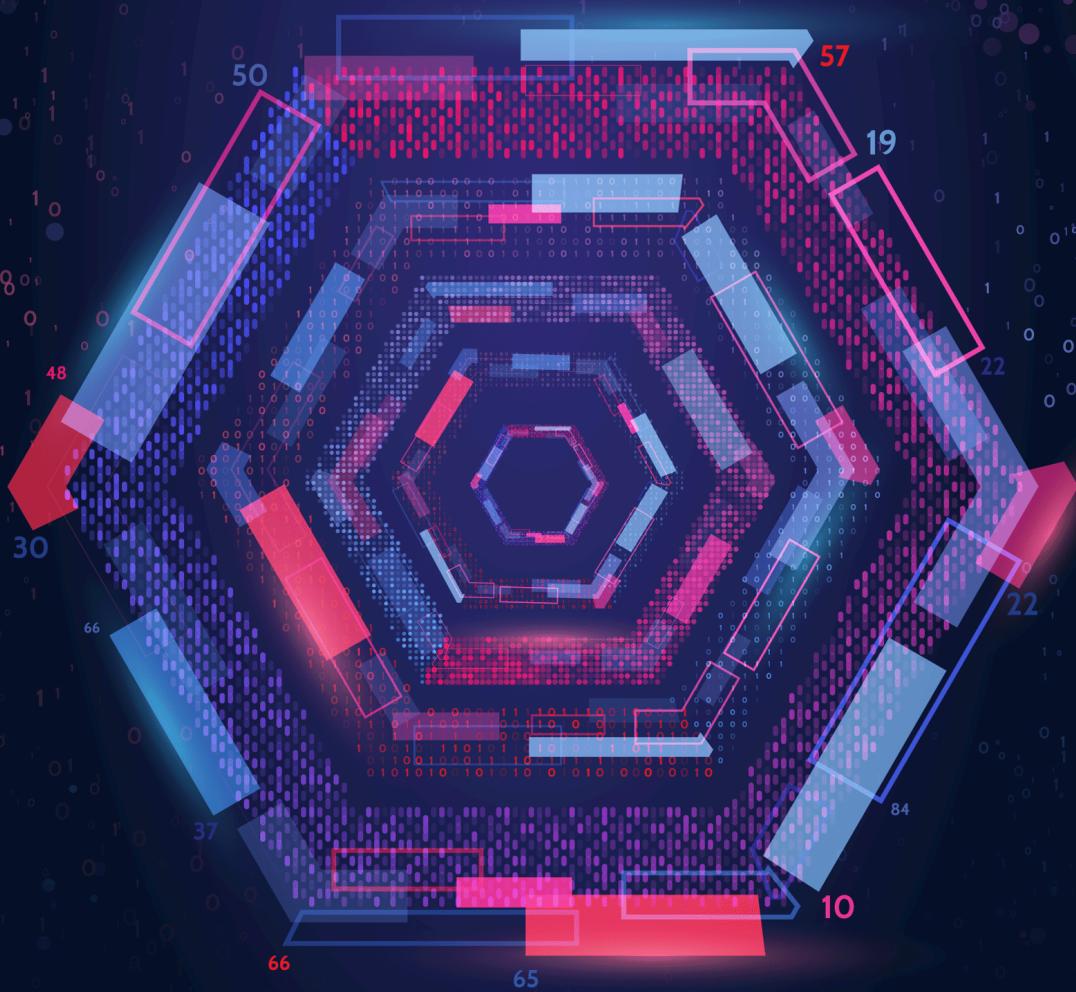


시작의 마흔에 뛰어들기



알렉산더 슈베츠

디자인 패턴에 뛰어들기

v2023-1.4

강민수님이 구매함
naman79@naver.com (#177637)

저작권에 대한 몇 마디

안녕하세요! 제 이름은 알렉산더 슈베츠입니다. 저는 『디자인 패턴에 뛰어들기¹』라는 책과 『리팩토링에 뛰어들기²』라는 온라인 교육 과정의 저자입니다.



이 책은 개인적인 용도로만 사용할 수 있습니다. 가족 이외의 제삼자와 공유하지 마세요. 친구나 동료와 책을 공유하고 싶다면 새 책을 사서 보내세요. 또 팀 전체 또는 회사 전체에 대한 사이트 사용권을 구매할 수도 있습니다.

제 책과 교육 과정의 판매로 얻은 모든 수익은 [Refactoring.Guru](#)의 개발에 사용됩니다. 판매되는 각 사본은 프로젝트를 지속하는 데 엄청난 도움이 되며 새 책의 출시를 가속합니다.

© 알렉산더 슈베츠, Refactoring.Guru, 2022

✉ support@refactoring.guru

▣ 삽화: 드미트리 자르트

▣ 옮긴이: 황우진 (Woo J. Hwang)

✎ 편집: [Alconost](#)

-
1. 『디자인 패턴에 뛰어들기』 :
<https://refactoring.guru/ko/design-patterns/book>
 2. 『리팩토링에 뛰어들기』 :
<https://refactoring.guru/ko/refactoring/course>

제 아내 마리아에게 이 책을 바칩니다. 그녀가 아니었다면
저는 아마 30년 후에 이 책을 완성했을 것입니다.

목차

목차	4
이 책을 읽는 방법	6
객체 지향 프로그래밍 소개	7
OOP의 기초	8
OOP의 기둥들	13
객체 간의 관계	21
디자인 패턴 소개	27
디자인 패턴이란?	28
왜 패턴을 배워야 할까요?	33
소프트웨어 디자인 원칙들	34
좋은 디자인의 특징	35
디자인 원칙들	40
▣ 변화하는 내용을 캡슐화하세요	41
▣ 구현이 아닌 인터페이스에 대해 프로그래밍하세요	46
▣ 상속보다 합성을 사용하세요	51
SOLID 원칙들	55
▣ 단일 책임 원칙	56
▣ 개방/폐쇄 원칙	58
▣ 리스코프 치환 원칙	62
▣ 인터페이스 분리 원칙	69
▣ 의존관계 역전 원칙	72

디자인 패턴 목록	76
생성 디자인 패턴	77
▣ 팩토리 메서드	79
▣ 추상 팩토리	96
▣ 빌더	112
▣ 프로토타입	131
▣ 싱글턴	147
구조 패턴	157
▣ 어댑터	160
▣ 브리지	175
▣ 복합체	191
▣ 데코레이터	206
▣ 퍼사드	226
▣ 플라이웨이트	237
▣ 프록시	252
행동 디자인 패턴	266
▣ 책임 연쇄	270
▣ 커맨드	290
▣ 반복자	311
▣ 중재자	327
▣ 멘토	343
▣ 옵서버	360
▣ 상태	377
▣ 전략	394
▣ 템플릿 메서드	408
▣ 비지터	422
결론	438

이 책을 읽는 방법

이 책에는 1994년 컴퓨터 공학자 사인조('Gang of Four' 또는 줄여서 GoF)가 공식화한 22가지 고전적인 디자인 패턴에 대한 설명이 포함되어 있습니다.

각 장은 특정 패턴을 탐구합니다. 따라서 당신은 책을 시작부터 끝까지 또는 관심 있는 패턴들을 선택하여 읽을 수 있습니다.

많은 패턴이 관련되어 있으므로 수많은 앱커들을 사용해 한 주제에서 다른 주제로 쉽게 이동할 수 있습니다. 각 장의 끝에는 현재 패턴과 관련된 다른 패턴들의 링크 목록이 있습니다. 아직 보지 못한 패턴의 이름이 보이면 계속 읽어 나가세요. 이 패턴은 곧 다음 장 중 하나에서 설명됩니다.

디자인 패턴은 모든 프로그래밍 언어에 보편적입니다. 따라서 이 책의 모든 코드 예시들은 특정 프로그래밍 언어에 제한되지 않는 의사 코드로 나타내었습니다.

패턴을 공부하기 전에 객체 지향 프로그래밍의 핵심 용어들을 복습할 수 있습니다. 이 장에서도 UML 다이어그램의 기초를 설명하는데, 이는 책에 수많은 다이어그램이 있으므로 유용합니다. 물론, 그 내용을 모두 알고 있다면 바로 패턴 학습 부분을 시작할 수 있습니다.

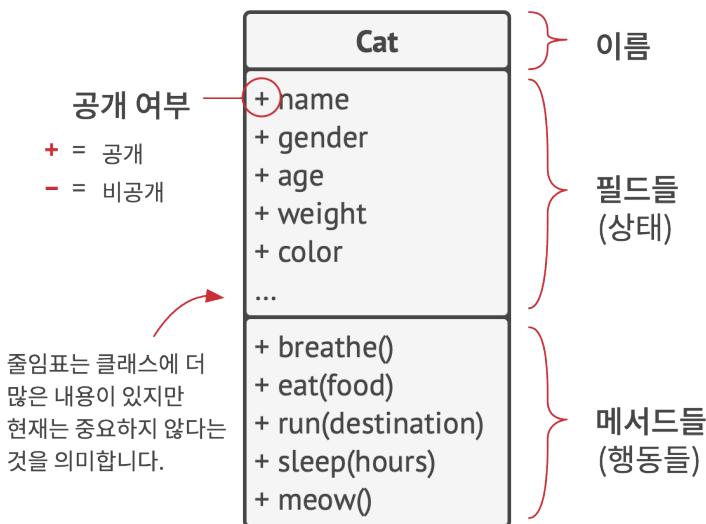
객체 지향 프로그래밍 소개

OOP의 기초

간략히 OOP라고도 불리는 **객체 지향 프로그래밍** (*Object Oriented Programming*)은 데이터 조각들 및 해당 데이터와 관련된 행동들을 **객체**라는 특수한 묶음으로 모은다는 개념에 기반한 이론적인 틀 또는 체계이며, 객체들은 **클래스**라고 하는 프로그래머가 정의한 '청사진'들의 집합으로 구성됩니다.

객체들, 클래스들

고양이 좋아하시나요? 다양한 고양이 예시를 사용하여 OOP라는 개념을 설명해 보겠습니다.



이것은 UML 클래스 다이어그램입니다. 이 책에서는 이러한 다이어그램을 많이 사용할 것입니다.

오스카라는 고양이가 있다고 가정해 봅시다. 오스카는 객체이며, `Cat` (고양이) 클래스의 인스턴스입니다. 각 고양이는 이름, 성별, 나이, 체중, 색깔, 좋아하는 음식 같은 일반적인 속성들을 많이 갖고 있습니다. 이러한 속성들을 클래스의 필드들이라고 합니다.

이 책에서 저는 클래스 이름을 UML 다이어그램이나 코드에 쓰인 것처럼 영어로 언급할 겁니다. 하지만 때로는 글이 대화처럼 읽힐 수 있게 클래스 이름을 한국어로 번역하여 언급할 수도 있으니 유념해 주세요.

모든 고양이는 비슷하게 행동합니다. 숨을 쉬고, 먹고, 뛰고, 자고, 야옹 소리를 내며 웁니다. 이것들은 클래스의 메서드들입니다. 필드들과 메서드들을 통틀어 해당 클래스의 멤버들이라고 부릅니다.

객체의 필드들의 내부에 저장된 데이터는 종종 상태라고 불리며, 객체의 모든 메서드들은 객체의 행동들을 정의합니다.

당신의 친구의 고양이인 루나 역시 `Cat` (고양이) 클래스의 인스턴스입니다. 루나는 오스카와 같은 속성의 집합을 가지고 있습니다. 이 둘의 차이점은 그들이 가진 이러한 속성들의 값이

다르다는 것입니다. 예를 들어 루나의 성별은 여성이고 색깔이 다르며 오스카보다 몸무게가 적습니다.



오스카: Cat

```
name      = "오스카"
sex       = "수컷"
age       = 3
weight    = 7
color     = 갈색
texture   = 줄무늬
```

루나: Cat

```
name      = "루나"
sex       = "암컷"
age       = 2
weight    = 5
color     = 회색
texture   = 무늬 없음
```

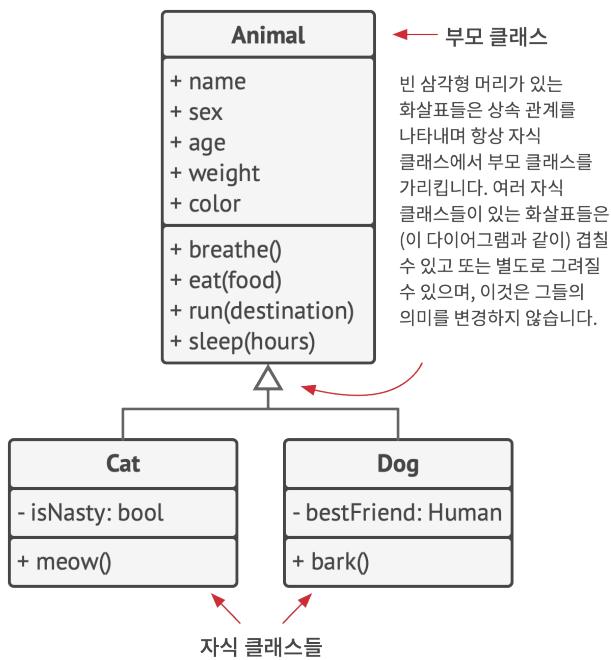
객체들은 클래스들의 인스턴스들입니다.

따라서 클래스는 객체들의 구조를 정의하는 청사진과 비슷하며, 객체들은 이 클래스의 구상 인스턴스들입니다.

클래스 계층구조들

한 클래스에 관해 이야기할 때는 모든 것이 간단하고 명확합니다. 그러나 대부분 실제 프로그램에는 하나 이상의 클래스가 포함되어 있습니다. 이러한 클래스 중 일부는 **클래스 계층구조**로 구성될 수 있으며, 지금부터는 그것이 무엇을 의미하는지 살펴보겠습니다.

이웃에 이름이 '파이도'인 개가 있다고 가정해 봅시다. 개와 고양이는 공통점이 많습니다. 이름, 성별, 나이, 색깔은 개와 고양이 모두의 속성입니다. 개는 고양이처럼 숨을 쉬고, 잠을 자고, 달릴 수도 있습니다. 그렇게 우리도 기초 Animal (동물) 클래스를 정의한 뒤 동물들의 일반적인 속성과 행동을 나열할 수 있을 것 같습니다.

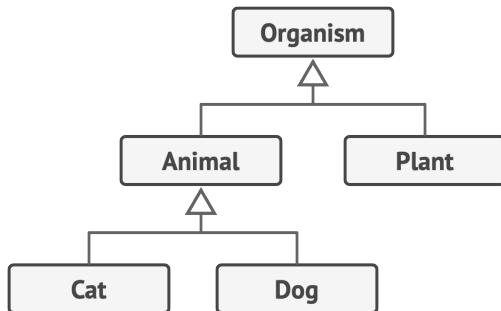


클래스 계층구조의 UML 다이어그램. 이 다이어그램의 모든 클래스는 Animal (동물) 클래스 계층구조의 일부입니다.

방금 정의한 것과 같은 상위 클래스를 **부모 클래스**라고 합니다.
그 하위에 있는 클래스들을 **자식 클래스들**이라고 합니다. 자식
클래스들은 부모로부터 상태와 행동들을 상속받고, 그중에서

부모와 무언가 다른 것들만을 정의합니다. 그래서 `Cat` (고양이) 클래스에는 `meow` (야옹) 메서드가 있고 `Dog` (강아지) 클래스에는 `bark` (멍멍 짖는) 메서드가 있게 됩니다.

연관된 비즈니스 요구사항이 있다면, 여기서 더 나아가 `Organisms` (모든 생명체들)라는 더욱 일반적인 클래스를 추출할 수도 있습니다. 이 클래스는 `Animals` (동물들) 및 `Plants` (식물들)에 대한 부모 클래스가 될 것입니다. 이런 클래스들의 피라미드가 바로 **계층구조**입니다. 이러한 계층구조에서 `Cat` (고양이) 클래스는 `Animal` (동물)과 `Organism` (생명체) 클래스 양쪽의 모든 내용을 상속받습니다.



만약 클래스들의 내용보다 그들 사이의 관계들을 표시하는 것이 더 중요한 경우 UML 다이어그램의 클래스들을 단순화할 수 있습니다.

자식 클래스들은 부모 클래스들에서 상속한 메서드들의 행동을 오버라이드할 수 있습니다. 또 자식 클래스들은 디폴트 행동들을 완전히 대체하거나 몇 가지 행동들을 추가하여 그 기능들을 향상시킬 수 있습니다.

OOP의 기둥들

객체 지향 프로그래밍(OOP)은 OOP를 다른 프로그래밍 틀 또는 체계들과 차별화하는 4가지 기둥, 즉 기본 개념들을 기반으로 합니다.

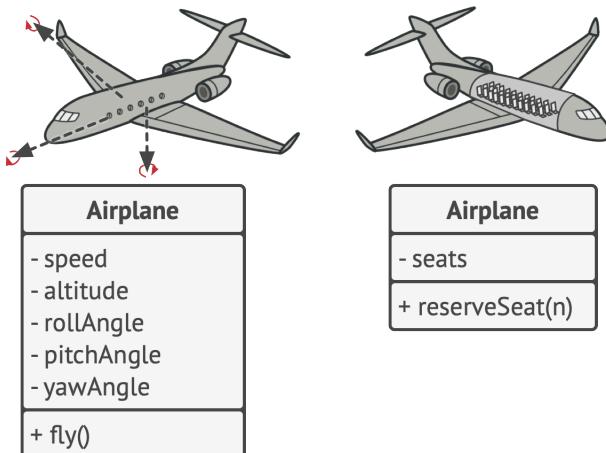


추상화

대부분의 경우 OOP를 사용하여 프로그램을 만들 때는 실생활에 존재하는 객체들을 기반으로 프로그램의 객체들을 형성합니다. 그러나 프로그램의 객체들은 실제 원본 객체들의 속성을 100% 정확하게 나타내지 않고, 대부분 그럴 필요도 없습니다. 대신 프로그램의 객체들은 특정 맥락에서만 실제 객체들의 속성들과 행동들을 모델링하고 나머지들은 무시합니다.

예를 들어 `Airplane` (비행기) 클래스는 비행 시뮬레이터와 항공 좌석 예약 앱 모두에 존재할 수 있습니다. 그러나

비행시뮬레이터가 실제 비행과 관련된 세부 정보들을 담고 있다면, 항공 좌석 예약 앱에서 사람들이 신경쓰는 건 좌석 배치도와 예약 가능한 좌석들 같은 정보들뿐입니다.



같은 실제 객체에 대한 다른 모델들.

추상화는 맥락에 따라 핵심적인 개념 또는 기능들로 제한되는 실제 객체 또는 현상의 모델이며, 이 맥락과 관련된 모든 세부 정보는 높은 정확도로 나타내고 나머지는 모두 생략합니다.

캡슐화

자동차 엔진을 시동하려면 버튼을 누르거나 키를 돌리기만 하면 됩니다. 후드 아래에 있는 전선들을 연결하고, 크랭크축과 실린더를 회전시켜 엔진의 전원 사이클을 시작할 필요가 전혀 없습니다. 왜냐하면 이러한 세세한 작업들은 자동차의 후드 아래에 숨겨져 있기 때문입니다. 운전자에게 있는 건 시동 스위치,

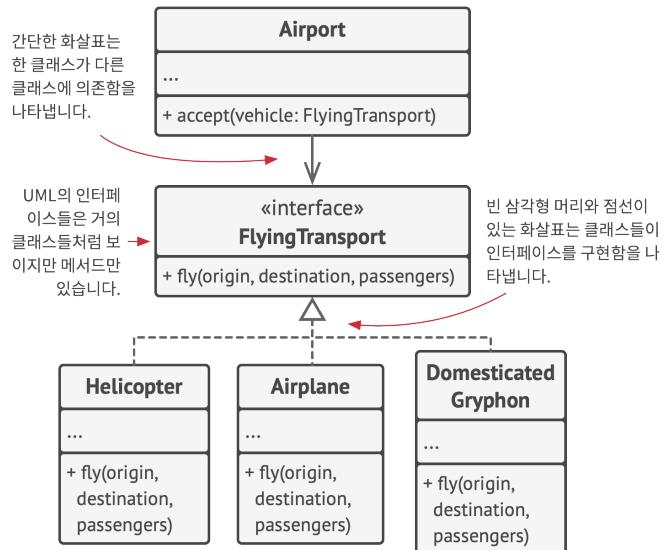
핸들, 그리고 몇 개의 페달이라는 단순한 인터페이스가 전부입니다. 이것은 각 객체가 **인터페이스**를 갖는 방식을 설명해줍니다. 인터페이스는 다른 객체와 상호작용할 수 있는 객체의 공개된 부분입니다.

캡슐화는 객체가 그 상태와 행동의 일부를 다른 객체들로부터 숨기고 나머지 프로그램에는 제한된 인터페이스만 노출할 수 있는 기능입니다.

무언가를 캡슐화한다는 것은 그것을 `private` (비공개)로 만든다는 뜻으로, 그러면 그 무언가를 자신의 클래스의 메서드 내에서만 접근할 수 있습니다. 조금 덜 제한적인 `protected` (보호된) 접근제한자라는 것도 있는데, 이것은 클래스의 멤버를 자식 클래스들에서도 사용할 수 있게 해줍니다.

대부분의 프로그래밍 언어의 인터페이스들, 추상 클래스들, 그리고 추상 메서드들은 추상화 및 캡슐화 개념들에 기반을 둡니다. 현대 OOP 언어들에서의 인터페이스 메커니즘 (일반적으로 `interface` 또는 `protocol` 키워드로 선언됨)은 객체 간의 상호 작용에 대한 계약을 정의할 수 있도록 합니다. 이는 인터페이스들이 객체들의 행동에만 관심을 두는 이유 중 하나이자, 당신이 인터페이스에서 필드를 선언할 수 없는 이유입니다.

인터페이스라는 단어는 객체의 공개된 부분을 나타냅니다. 동시에 대부분의 프로그래밍 언어에는 interface라는 유형도 있지요. 이 사실이 매우 혼란스럽다는 사실을 저는 잘 이해하고 있습니다.



인터페이스를 구현하는 여러 클래스의 UML 다이어그램.

`fly(origin, destination, passengers)` ((출발지, 목적지, 승객)을 인수로 받는 비행) 메서드가 있는 **FlyingTransport** (비행 운송 수단) 인터페이스가 있다고 가정해 봅시다. 항공 운송 시뮬레이터를 설계할 때 당신은 **Airport** (공항) 클래스를 **FlyingTransport** 인터페이스를 구현하는 객체들과만 작동하도록 제한할 수 있습니다. 이렇게 만들고 나면 **airplane** (비행기)이든,

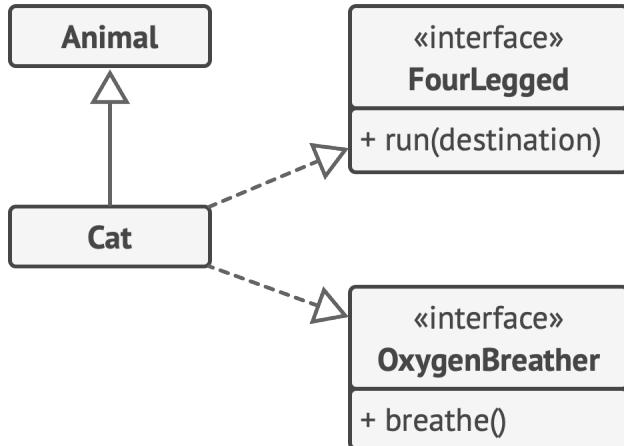
`Helicopter` (헬리콥터)든, 아니면 `DomesticatedGryphon` (집에서 키운 독수리사자)이든, 공항 객체에 전달된 어떤 객체라도 이러한 유형의 공항에서 이착륙할 수 있을 거라고 확신할 수 있습니다.

또 당신은 이러한 클래스들의 `fly` 메서드에 대한 구현을 원하는 방식으로 변경할 수 있습니다. 메서드의 시그니처들이 인터페이스에 선언된 것들과 같게 유지되는 한 `Airport` 클래스의 모든 인스턴스는 당신의 비행 객체들과 잘 작동할 수 있습니다.

상속

상속은 기존 클래스들 위에 새 클래스들을 구축하는 기능입니다. 상속의 가장 큰 이점은 코드 재사용입니다. 기존 클래스와 약간 다른 클래스를 만들고 싶을 때 기존 코드를 복제할 필요가 없죠. 그 대신 기존(부모) 클래스를 확장한 후 부모 클래스의 필드들과 메서드들을 상속한 결과 자식 클래스에 필요한 추가 기능들을 추가하면 됩니다.

상속을 사용하면 결과적으로 자식 클래스들이 부모 클래스와 같은 인터페이스를 갖게 됩니다. 어떤 메서드가 부모 클래스에서 선언되었다면 자식 클래스에서 그 메서드를 숨길 수 없습니다. 또한 자식 클래스들에 어울리지 않는 추상 메서드들을 포함하여 모든 추상 메서드들을 구현해야 합니다.



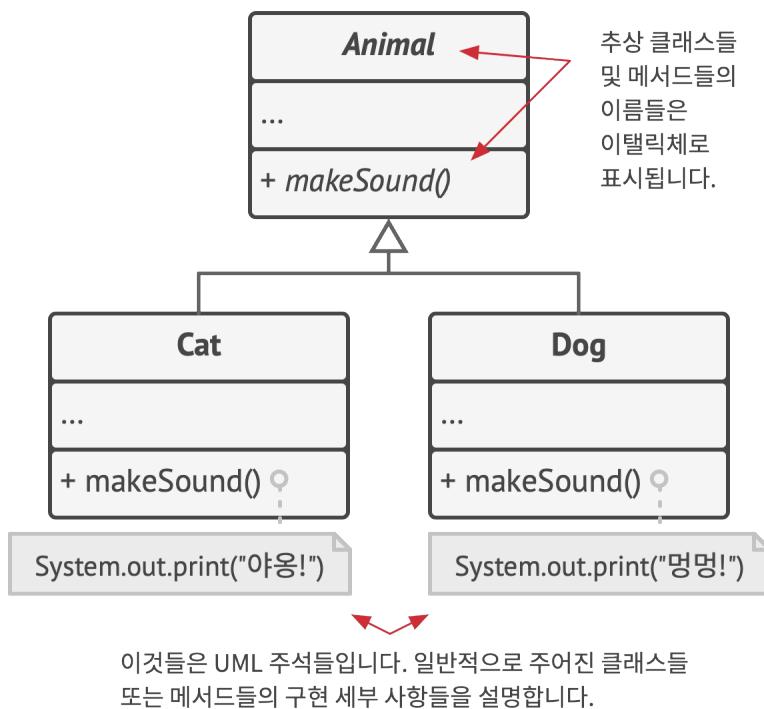
단일 클래스의 확장 vs. 여러 인터페이스의 동시 구현을 표현한
UML 다이어그램.

대부분의 프로그래밍 언어에서 자식 클래스는 하나의 부모 클래스만 확장할 수 있습니다. 반면에 모든 클래스가 동시에 여러 인터페이스를 구현할 수 있죠. 하지만 앞서 언급했듯이 부모 클래스가 인터페이스를 구현한다면 모든 자식 클래스들 또한 그 인터페이스를 구현해야 합니다.

다형성

이제 동물에 대한 예시를 살펴봅시다. 대부분의 **Animals** (동물들)는 소리를 낼 수 있습니다. 따라서 우리는 모든 자식 클래스들이 기초 `makeSound` (소리내기) 메서드를 오버라이드해야 각 자식 클래스가 그에 해당하는 동물의 소리를 올바르게 낼 수 있다고 예상할 수 있습니다. 그러므로 우리는 이 메서드를 바로 **추상**으로 선언할 수 있습니다. 이렇게 하면

부모 클래스에서 이 메서드의 디폴트 구현을 생략할 수 있습니다.
하지만 모든 자식 클래스들은 강제적으로 이 메서드를 각자 구현해야 합니다.



이제 당신이 큰 가방에 여러 고양이와 개들을 넣었다고 가정해 봅시다. 그런 다음 눈을 감은 상태에서 가방에서 동물들을 하나씩 꺼내는 겁니다. 가방에서 동물을 꺼낸 직후 당신은 그 동물이 어떤 동물인지 확실히 모릅니다. 하지만 그 동물을 힘껏 껴안아 본다면 이 동물은 그 구상 클래스에 따라 특정한 소리를 낼 것입니다.

```
1 bag = [new Cat(), new Dog()];
2
3 foreach (Animal a : bag)
4     a.makeSound()
5
6 // 야옹!
7 // 명명!
```

프로그램은 `a` 변수 안에 포함된 객체의 구상 유형을 알지 못합니다. 하지만 **다형성**이라는 특별한 메커니즘 덕분에, 프로그램은 그 메서드가 실행되어 적절한 행동들을 실행하는 객체의 자식 클래스를 추적할 수 있습니다.

다형성은 객체의 실제 클래스를 감지하고 해당 객체의 구현을 현재 맥락에서 이것의 실제 유형을 알 수 없는 경우에도 호출할 수 있는 프로그램의 기능입니다.

다형성은 객체가 다른 무언가인 척 '가장'을 할 수 있는 기능이라고도 생각할 수 있습니다. 일반적으로는 객체가 확장하는 클래스 또는 구현하는 인터페이스인 척 가장합니다. 위 예시에서는 가방에 든 개와 고양이가 일반적인 동물인 척 가장하고 있었습니다.

객체 간의 관계

전에 설명한 상속 및 구현 외에도 객체들 사이에는 다른 유형의 관계들이 있습니다.

의존성



의존성 UML. 교수는 강의 자료에 의존합니다.

의존성은 클래스 간의 가장 기본적이고 약한 유형의 관계입니다. 한 클래스의 정의를 일부 변경했을 때 다른 클래스가 변경되는 경우 두 클래스 사이에 의존성이 있다고 할 수 있습니다. 의존성은 일반적으로 코드에서 구상 클래스 이름들을 사용할 때 만들어집니다. (예: 메서드 시그니처 유형들을 지정할 때, 생성자 호출들을 통해 객체들을 인스턴스화할 때 등). 당신의 코드를 구상 클래스가 아닌 인터페이스나 추상 클래스에 의존하게 만들면 의존성을 약화할 수 있습니다.

일반적으로 UML 다이어그램은 모든 의존 관계를 표시하지 않습니다. 왜냐하면 실제 코드에는 너무나도 많은 의존 관계들이 있기 때문입니다. 따라서 의존 관계로 다이어그램을 복잡하게

만드는 대신, 나타내고자 하는 아이디어에 중요한 관계들만 매우 선별적으로 표시해야 합니다.

연관 관계



연관관계 UML. 교수는 학생들과 서로 대화합니다.

연관은 한 객체가 다른 객체를 사용하거나 이 두 객체가 상호 작용하는 관계입니다. UML 다이어그램에서 연관 관계는 한 객체에서 그 객체가 사용하는 객체로 향하는 간단한 화살표로 표시됩니다. 참고로 양방향 연관 관계는 완전히 정상적이며, 이 경우 화살의 양 끝에 화살표가 있습니다. 연관은 특수한 종류의 의존관계로 간주할 수 있으며, 이때 객체는 언제나 상호작용하는 객체에 접근할 수 있습니다. 반면 단순한 의존관계에서는 객체 사이의 영구적인 연결이 만들어지지 않습니다.

일반적으로 다른 객체가 포함된 필드를 나타낼 때는 연관관계를 사용합니다. 이 필드는 두 객체 간의 링크 역할을 합니다. 하지만 늘 필드일 필요는 없습니다. 연관관계는 일부 객체를 반환하는 메서드로 나타낼 수도 있습니다. 그렇지 않으면 인터페이스에 필드가 없기 때문에 인터페이스 사이에서 연관관계를 사용할 수 없을 것입니다.

연관 관계와 의존 관계의 차이를 확실히 짚고 넘어갈 수 있게 다른 예를 하나 더 살펴봅시다. `Professor` (교수) 클래스가 있다고 가정합시다.

```

1 class Professor is
2   field Student student
3   // ...
4   method teach(Course c) is
5     // ...
6     this.student.remember(c.getKnowledge())

```

이제 `teach` 메서드를 살펴보세요. 이 메서드는 `Course` (수업 과정) 클래스를 인수로 받으며, 이 인수는 메서드의 본문에서 사용됩니다. 누군가가 `getKnowledge` 메서드의 시그니처를 변경하면(예: 이름을 바꾼다든지 또는 필요한 매개변수를 추가한다든지 등) 코드가 더 이상 작동하지 않을 것입니다. 바로 이것이 `Professor` 클래스가 `Course` 클래스에 의존한다고 말할 수 있는 이유입니다.

이제 `student` 필드를 살펴보고 해당 필드가 `teach` 메서드에서 어떻게 사용되는지도 살펴보세요. `Student` 클래스는 `Professor` 클래스에 의존하는 dependency라고 확실히 말할 수 있습니다. 왜냐하면 `remember` 메서드의 시그니처가 변경되면 `Professor`의 코드가 깨지기 때문입니다. 그러나 `Professor`의 모든 메서드는 `student` 필드에 접근할 수 있으므로, `Student`

클래스는 Professor에 의존하는 dependency일 뿐만이 아니라 Professor와 연관 관계가 있기도 합니다.

집합 관계



집합 관계의 UML. 학과는 교수들을 포함합니다.

집합 관계는 여러 객체 간의 '일-대-다', '다-대-다' 또는 '전체-부분' 관계들을 나타내는 특수한 유형의 연관 관계입니다.

일반적으로 집합 관계에서 객체는 다른 객체들의 집합을 '가지며', 컨테이너 또는 컬렉션 역할을 합니다. 컴포넌트는 컨테이너 없이 존재할 수 있으며, 동시에 여러 컨테이너에 연결될 수 있습니다. UML에서 집합 관계는 컨테이너 끝에서는 빈 다이아몬드가 달려있고 컴포넌트를 가리키는 끝부분에서는 화살표가 달린 선으로 표시됩니다.

UML은 클래스들 간의 관계를 나타냅니다. 이것은 디어그램에서의 각 사물이 하나의 '블록'만으로 표시되더라도 대학 객체가 여러 학과로 구성될 수 있음을 의미합니다. UML 표기법에서는 관계의 양쪽에 수량을 표시낼 수 있지만 이 수량이 맥락상 명확하면 생략해도 됩니다.

합성 관계



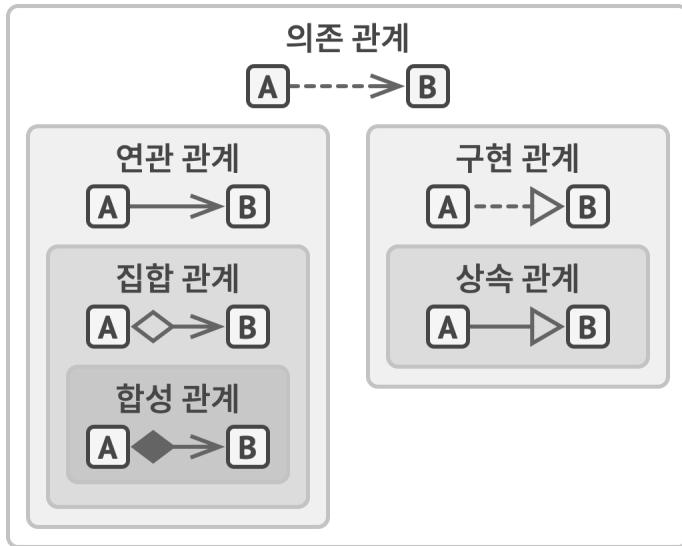
합성 관계 UML. 대학은 많은 학과들을 포함합니다.

합성 관계는 특정 유형의 집합 관계로, 여기서 객체는 다른 객체의 하나 또는 그 이상의 인스턴스로 구성됩니다. 합성 관계와 다른 관계들의 차이점은 컴포넌트가 컨테이너의 일부로만 존재할 수 있다는 것입니다. UML에서 합성 관계는 집합 관계와 동일하게 표시되지만, 화살표에 속이 채워진 다이아몬드가 추가됩니다.

많은 사람이 '합성'이라는 용어를 합성과 집합 관계 모두를 뜻하는 상황에서 사용하곤 합니다. 그 가장 악명 높은 예시가 바로 '상속보다 합성을 선택하십시오'라는 유명한 원칙입니다. 사람들이 둘 사이의 차이점에 대해 무지해서가 아니라 합성을 뜻하는 `composition` (구성)이라는 단어(예: '객체 구성')가 영어에서는 더 자연스럽게 들리기 때문입니다.

큰 그림

이제까지 객체 간의 모든 관계에 대해 살펴보았으므로 이 관계들이 어떻게 연결되었는지 살펴봅시다. 이 설명이 '집합 관계와 합성 관계의 차이점은 무엇입니까' 또는 '상속 관계는 의존 관계의 일종입니까'와 같은 질문들에 답이 되었기를 바랍니다.



객체들과 클래스들 간의 관계들: 가장 약한 관계에서부터 가장 강한 관계까지.

- **의존 관계:** 클래스 A는 클래스 B가 변경될 때 영향을 받을 수 있습니다.
- **연관 관계:** 객체 A가 객체 B에 대해 알고 있습니다. 클래스 A는 B에 의존합니다.
- **집합 관계:** 객체 A가 객체 B에 대해 알고 있으며 B로 구성됩니다. 클래스 A는 B에 의존합니다.
- **합성 관계:** 객체 A가 객체 B에 대해 알고, B로 구성되며 B의 수명 주기를 관리합니다. 클래스 A는 B에 의존합니다.
- **구현 관계:** 클래스 A가 인터페이스 B에 선언된 메서드를 정의합니다. 객체 A는 B처럼 처리될 수 있습니다. 클래스 A는 B에 의존합니다.
- **상속 관계:** 클래스 A가 클래스 B의 인터페이스 및 구현 관계를 상속하지만 이를 확장할 수 있습니다. 객체 A는 B처럼 처리될 수 있습니다. 클래스 A는 B에 의존합니다.

디자인 패턴 소개

디자인 패턴이란?

디자인 패턴은 소프트웨어 디자인 과정에서 자주 발생하는 문제들에 대한 전형적인 해결책입니다. 이는 코드에서 반복되는 디자인 문제들을 해결하기 위해 맞춤화할 수 있는 미리 만들어진 청사진과 비슷합니다.

표준화된 라이브러리들이나 함수들을 코드에 복사해 사용하는 것처럼 패턴들을 붙여넣기식으로 사용할 수 없습니다. 패턴은 재사용할 수 있는 코드 조각이 아니라 특정 문제를 해결하는 방식을 알려주는 일반적인 개념들입니다. 당신은 패턴의 세부 개념들을 적용하여 당신의 프로그램에 맞는 해결책을 구현할 수 있습니다.

패턴은 알고리즘과 자주 혼동됩니다. 왜냐하면 두 개념 모두 알려진 문제에 대한 일반적인 해결책을 설명하기 때문입니다. 알고리즘은 어떤 목표를 달성하기 위해 따라야 할 명확한 일련의 절차를 정의하지만, 패턴은 해결책에 대한 더 상위 수준의 설명입니다. 예를 들어 같은 패턴을 두 개의 다른 프로그램에 적용하면 두 프로그램의 코드는 다를 것입니다.

알고리즘은 요리법에 비유할 수 있지만 패턴은 요리법이 아닌 청사진에 더 가깝습니다. 알고리즘과 요리법 둘 다 목표를 달성하기 위한 명확한 단계들이 제시되어 있습니다. 반면에

청사진은 결과와 기능들은 제시하나 구현 단계 및 순서는 사용자가 결정합니다.

☰ 패턴은 무엇으로 구성되어 있나요?

많은 상황에서 독자들이 패턴을 재현할 수 있도록 대부분의 패턴을 매우 형식적으로 설명했습니다. 패턴 설명에 일반적으로 표시되는 섹션들은 다음과 같습니다.

- **패턴의 의도** 섹션에서는 문제와 해결책을 간략하게 설명했습니다.
- **동기** 섹션에서는 문제와 패턴이 가능하게 하는 해결책을 추가 설명했습니다.
- 클래스의 **구조** 섹션에서는 패턴의 각 부분과 이러한 부분들이 어떻게 연관되어 있는지를 보여주었습니다.
- **코드 예시** 섹션에서는 여러 인기 있는 프로그래밍 언어들로 된 코드 예시를 제공하여 독자들이 패턴 뒤의 아이디어를 이해하기 쉽도록 했습니다.

일부 패턴 섹션에서는 패턴의 적용, 구현 단계 및 다른 패턴과의 관계와 같은 유용한 세부 정보들도 설명했습니다.

☰ 패턴의 분류

디자인 패턴은 복잡성, 상세도 및 설계 중인 전체 시스템에 대한 적용 범위에 따라 분류됩니다. 저는 도로 건설에 비유하는 걸

좋아합니다. 교차로를 더 안전하게 만들기 위해 신호등을 설치하거나 보행자를 위한 지하도가 있는 전체 다층 인터체인지지를 구축하는 작업에 비유할 수 있습니다.

가장 기본적인 하위 수준 패턴을 이디엄이라고 합니다. 일반적으로 이디엄은 하나의 프로그래밍 언어에만 해당합니다.

가장 보편적인 상위 수준 패턴은 아키텍처 패턴입니다. 개발자들은 거의 모든 언어로 이러한 패턴을 구현할 수 있습니다. 다른 패턴들과 달리 아키텍처 패턴은 애플리케이션 전체의 구조 (아키텍처)를 설계하는 데 사용할 수 있습니다.

또한 모든 패턴은 패턴의 의도 또는 목적에 따라 분류할 수 있습니다. 이 책에서는 패턴의 주요 세 가지 그룹에 대해 다룹니다.

- **생성 패턴**은 기존 코드를 재활용하고 유연성을 증가시키는 객체 생성 메커니즘을 제공합니다.
- **구조 패턴**은 구조를 유연하고 효율적으로 유지하면서 객체와 클래스를 더 큰 구조로 조합하는 방법을 설명합니다.
- **행동 패턴**은 객체 간의 효과적인 의사소통과 책임 할당을 처리합니다.

▣ 누가 패턴을 발명했나요?

좋지만 아주 정확하진 않은 질문입니다. 디자인 패턴은 모호하고 복잡한 개념이 아니라 그 반대입니다. 패턴은 객체 지향 설계의 일반적인 문제에 대한 전형적인 해결책입니다. 한 해결책이 다양한 프로젝트에서 계속 반복되면 결국 누군가가 이름을 붙이고 해결책을 자세히 설명합니다. 이것이 기본적으로 패턴이 발견되는 방법입니다.

패턴이라는 개념은 『패턴 랭귀지: 도시 건축 시공¹』 (크리스토퍼 알렉산더)에서 처음으로 설명되었습니다. 이 책은 도시 환경을 설계하기 위한 '언어'를 설명하며, 이 언어의 단위가 패턴입니다. 패턴으로 창문이 얼마나 높아야 하는지, 건물이 몇 층이어야 하는지, 근처 녹지의 면적이 얼마나 넓어야 되는지 등을 설명할 수 있습니다.

이 개념은 에릭 감마, 존 블리시디스, 랄프 존슨, 리처드 헬름이라는 4명의 작가에 의해 정리되었습니다. 1994년에 이들은 『디자인 패턴: 재사용성을 지닌 객체지향 소프트웨어의 핵심 요소²』라는 책을 발간하였고, 거기서 디자인 패턴의 개념을 프로그래밍에 적용하였습니다. 이 책은 23가지의 객체 지향 디자인 관련 문제를 해결하는 패턴을 소개했으며 빠르게 베스트

-
1. 『패턴 랭귀지: 도시 건축 시공』 : <https://refactoring.guru/ko/pattern-language-book>
 2. 『디자인 패턴: 재사용성을 지닌 객체지향 소프트웨어의 핵심 요소』 : <https://refactoring.guru/gof-book>

셀러가 되었습니다. 너무 긴 제목 때문에 사람들은 이 책을 '4인방 (Gang of Four)이 쓴 책'이라고 부르기 시작했으며 곧 간단히 'GoF 책'으로 줄여 부르게 되었습니다.

그 이후로 수많은 다른 객체 지향 패턴이 발견되었습니다. '패턴 접근법'이 다른 프로그래밍 분야에서도 인기를 얻으며 이제 객체 지향 디자인 외부에도 많은 패턴이 존재합니다.

왜 패턴을 배워야 할까요?

현실은 당신이 패턴에 대해 아무것도 알지 못해도 수년 동안 프로그래머로 일할 수 있다는 것입니다. 실제로 많은 프로그래머가 패턴에 대한 아무런 지식 없이 업무를 수행합니다. 또 자신도 모르는 사이에 일부 패턴들을 구현하고 있을 수도 있습니다. 그럼에도 왜 패턴을 배워야 하는지, 그 이유들을 정리해 보겠습니다.

- 다양한 디자인 패턴은 소프트웨어 디자인의 일반적인 문제들에 대해 **시도되고 검증된** 해결책들을 모은 것입니다. 이러한 문제들을 다루지 않더라도 패턴을 알고 있으면 여전히 쓸모가 있는데, 그 이유는 패턴을 배우게 되면 객체 지향 디자인의 원칙들을 사용해 많은 종류의 문제를 해결하는 방법들을 배울 수 있기 때문입니다.
- 디자인 패턴은 당신과 당신의 팀원들이 더 효율적으로 의사소통하는 데 사용할 수 있는 공통 언어를 정의합니다. 예를 들어서 당신의 팀이 디자인 패턴을 이해하면 업무 처리 중 당신이 '그 문제를 위해서는 그냥 싱글턴을 사용하세요'라고 말하면 모두가 당신이 무엇을 뜻했는지 이해할 수 있습니다. 싱글턴 패턴에 포함된 개념들은 설명할 필요도 없죠.

소프트웨어 디자인 원칙들

좋은 디자인의 특징

실제 패턴에 대해 논의하기 전에 소프트웨어 아키텍처를 설계하는 과정과 그 과정에서 목표로 삼거나 피해야 할 사항들에 대해 논의해 보겠습니다.

⚒️ 코드 재사용

모든 소프트웨어 제품을 개발할 때 가장 중요한 두 가지 지표는 비용과 시간입니다. 개발 시간이 짧으면 경쟁자들보다 일찍 시장에 진입할 수 있으며, 개발 비용이 낮아지면 마케팅에 더 많은 자금을 투자하여 더 광범위한 잠재 고객들에 접근할 수 있습니다.

코드 재사용은 개발 비용을 줄이는 가장 일반적인 방법의 하나입니다. 코드를 재사용하는 의도는 상당히 명백합니다. 무언가를 계속 처음부터 다시 개발하지 말고, 기존 코드를 새 프로젝트에 다시 사용하자는 것이죠.

이 개념은 이론상으로는 훌륭해 보입니다. 그러나 새로운 상황에서 기존 코드를 작동하게 하는 것은 쉽지 않습니다. 왜냐하면 컴포넌트 간의 단단한 결합, 인터페이스 대신 구상 클래스들에 대한 의존 관계, 하드코딩 된 작업과 같은 여러 가지 요인들이 코드의 유연성을 감소시키고 재사용을 어렵게 만들기 때문입니다.

디자인 패턴을 사용하는 것은 소프트웨어 컴포넌트들의 유연성을 높이고 재사용하기 쉽게 만드는 한 가지 방법입니다. 그러나 이 방법은 때때로 컴포넌트들을 더 복잡하게 만듭니다.

다음은 디자인 패턴의 선구자 중 한 명인 에릭 감마¹의 코드 재사용에서의 디자인 패턴의 역할에 대한 견해입니다.

“

저는 재사용을 세 가지 수준으로 이해합니다.

가장 낮은 수준에서는 클래스들을 재사용합니다. 그 예로는 클래스 라이브러리, 컨테이너 및 컨테이너/반복자와 같은 어떤 클래스 '팀'이 있습니다.

반면 프레임워크들은 최고 수준에 있으며, 이들은 당신의 디자인 결정들을 정제하려고 열심히 노력합니다. 프레임워크들은 문제를 해결하기 위한 핵심 추상화들을 식별한 후, 해당 추상화들을 클래스들로 표현하고 그들 사이의 관계들을 정의합니다. 예를 들어 JUnit은 작은 프레임워크이며, 프레임워크의 'Hello, world'라고 간주할 수 있습니다. 이 프레임워크에는 `Test`, `TestCase`, `TestSuite` 및 관계들이 정의되어 있습니다.

프레임워크는 일반적으로 단일 클래스보다 입자들이 큽니다. 또한 프레임워크에 연결할 때는 어딘가를 서브클래싱하여 연결합니다. 그들은 '전화하지 마, 전화할께'라는 이른바 헐리우드 원칙을 사용합니다. 프레임워크는 사용자 지정 행동을 정의할 수 있도록

-
1. 에릭 감마가 말하는 유연성과 재사용: <https://refactoring.guru/gamma-interview>

해주고, 당신이 어떤 작업을 수행할 차례가 되면 당신을 호출할 것입니다. JUnit도 마찬가지죠? JUnit도 당신을 위해 테스트를 실행하고 싶을 때는 당신을 호출하지만, 나머지는 프레임워크에서 작동합니다.

중간 수준의 재사용도 있습니다. 저는 패턴이 이 수준에 속한다고 생각합니다. 디자인 패턴은 프레임워크보다 더 작고 추상적입니다. 디자인 패턴은 몇 개의 클래스들이 서로 어떻게 관련되어 있으며, 상호 작용할 수 있는지에 대한 상세한 설명입니다. 클래스에서 패턴으로, 그리고 마지막으로 프레임워크로 이동할수록 재사용 수준이 높아집니다.

이 중간 수준 계층의 좋은 점은 패턴이 종종 프레임워크보다 덜 위험한 방식의 재사용을 제공한다는 점입니다. 프레임워크를 구축하는 것은 위험이 높을 뿐만 아니라 상당한 투자가 들어가는 일입니다. 패턴을 사용하면 구상 코드와 관계 없이 디자인 아이디어들과 개념들을 재사용할 수 있습니다.

”

확장성

변화는 프로그래머의 삶에서 유일하게 변하지 않는 것입니다.

- 윈도우용으로 비디오 게임을 출시했는데, 사람들은 이제 맥용 버전을 요구합니다.
- 그래픽 사용자 인터페이스 프레임워크에 네모난 버튼들을 만들었는데, 몇 개월 후에 둥근 버튼이 유행하게 되었습니다.

- 뛰어난 전자상거래 웹사이트 아키텍처를 디자인했는데, 불과 한 달 후 고객들이 전화 주문을 수락할 수 있는 기능을 요청합니다.

대부분 소프트웨어 개발자들은 이와 같은 상황들을 수십 번 이상 경험했을 것입니다. 이러한 상황들이 발생하는 데는 몇 가지 이유가 있습니다.

첫 번째 이유는 일단 문제를 해결하기 시작하면 문제를 더 잘 이해할 수 있기 때문입니다. 종종 개발자들은 앱의 첫 번째 버전의 개발을 끝마칠 때쯤에 문제의 여러 측면을 훨씬 더 잘 이해하기 때문에 처음부터 다시 개발하고 싶어 할 수 있습니다. 또, 개발자로서의 실력이 향상된 덕분에 작성한 코드가 쓰레기처럼 보일 수도 있습니다.

두 번째 이유는 통제할 수 없는 무언가가 변경되었기 때문입니다. 이는 많은 개발 팀들이 원래 아이디어에서 새로운 아이디어로 선회하는 이유입니다. 예를 들어 온라인 앱에서 플래시에 의존했던 개발자들은 브라우저들이 플래시에 대한 지원을 중단했을 때 코드를 재작성하거나 마이그레이션해야 했습니다.

세 번째 이유는 목표들이 변했기 때문입니다. 예를 들어 당신의 고객은 앱의 현재 버전에 만족했고 이제 원래 기획 단계 미팅에서 언급하지 않은 11개의 '작은' 변경을 수행하기를 원합니다. 사실 이러한 변경은 사소하지 않습니다. 당신의 첫 번째 버전이 훌륭하여 고객에게 더 많은 것들이 가능하다는 것을 보여준 것이죠.

긍정적인 측면도 있습니다. 누군가 당신의 앱에서 뭔가를 바꿔 달라고 요청한다면, 그건 누군가 당신의 앱에 여전히 관심이 있다는 뜻이니까요.

그러므로 모든 노련한 개발자들은 앱의 아키텍처를 설계할 때 미래에 변경들이 가능하게 하려고 노력합니다.

디자인 원칙들

좋은 소프트웨어 디자인이란 무엇일까요? 그리고 그건 어떻게 측정할까요? 또 이를 달성하기 위해서는 어떤 관행을 따라야 할까요? 아키텍처를 유연하고 안정적이며 이해하기 쉽게 만드는 방법은 무엇일까요?

훌륭한 질문들이나 불행히도 정답은 개발 중인 앱의 유형에 따라 다릅니다. 그래도 소프트웨어 설계에는 몇 가지 보편적인 원칙들이 있으며, 이러한 원칙들이 당신의 프로젝트에 대한 위 질문들에 답하는 데 도움이 될 수 있습니다. 이 책에 나열된 대부분의 디자인 패턴들은 이러한 원칙들을 기반으로 합니다.

변화하는 내용을 캡슐화하세요

당신의 앱에서 변경되는 부분들을 식별한 후 변하지 않는 부분들과 구분하세요.

이 원칙의 가장 큰 목적은 변화로 인해 발생하는 결과를 최소화하는 것입니다.

당신의 앱이 선박이고 변경 사항들은 물속에 대기하고 있는 끔찍한 지뢰들이라고 상상해 보세요. 지뢰에 걸리면 선박은 가라앉습니다.

이 사실을 알기에 당신은 선박의 선체를 독립된 구획으로 나눌 수 있습니다. 그러면 선박이 지뢰에 부딪쳐도 구획 하나만 안전하게 봉쇄하면 선박 전체는 여전히 떠 있게 됩니다.

동일한 방식으로 당신은 독립된 모듈에서 변경되는 프로그램의 일부를 따로 떼어내어 코드의 나머지 부분들을 역효과로부터 보호할 수 있습니다. 그렇게 하면 프로그램을 다시 작동하는 상태로 만들고, 변경 사항들을 구현하고 테스트하는 데 걸리는 시간을 줄일 수 있습니다. 변경 사항을 적용하는 데 드는 시간이 줄어들수록, 기능을 구현하는 데 더 많은 시간을 쓸 수 있을 것입니다.

메서드 수준에서의 캡슐화

당신이 전자 상거래 웹사이트를 개발하고 있다고 가정해 봅시다. 코드 어딘가에 세금을 포함한 주문의 총계를 계산하는 `getOrderTotal` 메서드가 있습니다.

우리는 앞으로 세금 관련 코드를 변경해야 할지도 모른다는 사실을 예측할 수 있습니다. 왜냐하면 세율은 당신이 거주하는 국가, 주, 도시에 따라 다르고, 시간이 흘러 새로운 법률이나 규정이 나와 실제 공식을 변경해야 할 수도 있기 때문입니다. 그러면 `getOrderTotal` 메서드를 아주 자주 변경해야 할 수도 있습니다. 그러나 이 메서드의 이름에서도 알 수 있듯이 해당 메서드는 세금이 계산되는 방법에는 아무 관심이 없습니다.

```

1 method getOrderTotal(order) is
2     total = 0
3     foreach item in order.lineItems
4         total += item.price * item.quantity
5
6     if (order.country == "US")
7         total += total * 0.07 // US sales tax
8     else if (order.country == "EU"):
9         total += total * 0.20 // European VAT
10
11    return total

```

수정 전: 세금 계산 코드가 메서드의 나머지 코드와 뒤섞여 있습니다.

당신은 이제 세금 계산 로직을 별도의 메서드로 추출하여 원래 메서드로부터 숨길 수 있습니다.

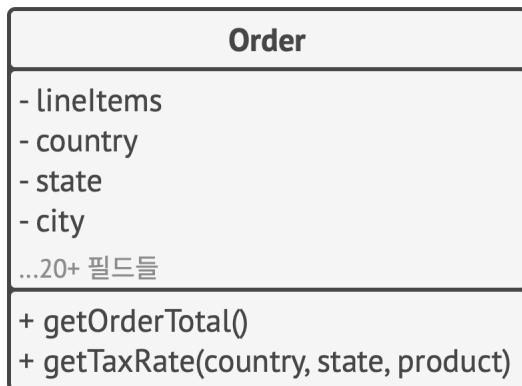
```
1  method getOrderTotal(order) is
2      total = 0
3      foreach item in order.lineItems
4          total += item.price * item.quantity
5
6      total += total * getTaxRate(order.country)
7
8      return total
9
10 method getTaxRate(country) is
11     if (country == "US")
12         return 0.07 // US sales tax
13     else if (country == "EU")
14         return 0.20 // European VAT
15     else
16         return 0
```

수정 후: 지정된 메서드를 호출함으로써 세율을 구할 수 있습니다.

이제 세금 관련 변경 사항들은 단일 메서드 내에서 격리됩니다. 또 세금 계산 로직이 너무 복잡해지면 그것을 별도의 클래스로 옮기기도 쉬워졌습니다.

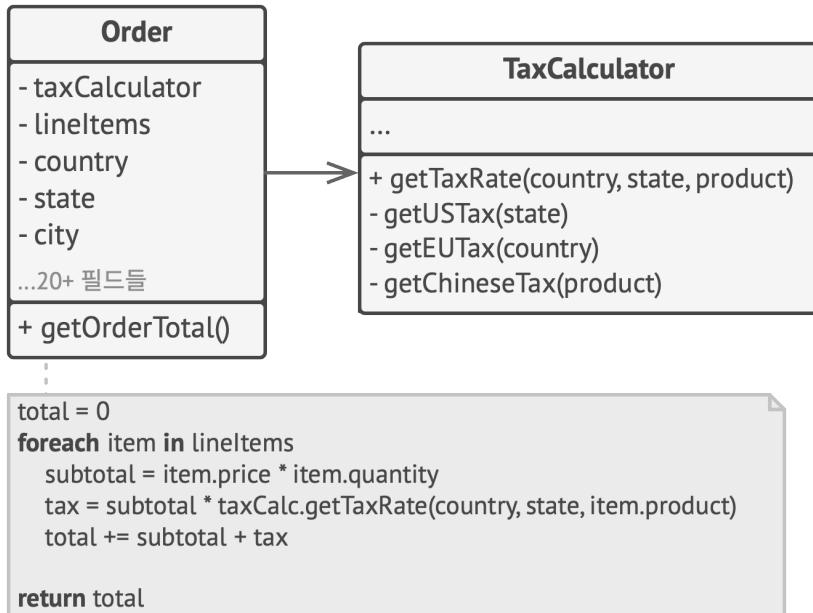
클래스 수준에서의 캡슐화

시간이 흐르면서 이전에는 간단한 작업을 수행했던 메서드에 점점 더 많은 책임이 추가될 수 있습니다. 이렇게 추가된 행동들은 고유의 도우미 필드와 메서드를 동반하곤 하는데, 이는 결국 이 모든 것을 포함하는 클래스의 기본적인 책임을 모호하게 만듭니다. 이 모든 것들을 새 클래스로 추출하면 코드는 훨씬 명확하고 간단해집니다.



수정 전: Order (주문) 클래스의 세금 계산.

Order 클래스의 객체들이 모든 세금 관련 작업을 해당 작업만 수행하는 특수 객체에 위임합니다.



수정 후: 세금 계산은 Order (주문) 클래스로부터 숨겨집니다.

구현이 아닌 인터페이스에 대해 프로그래밍하세요

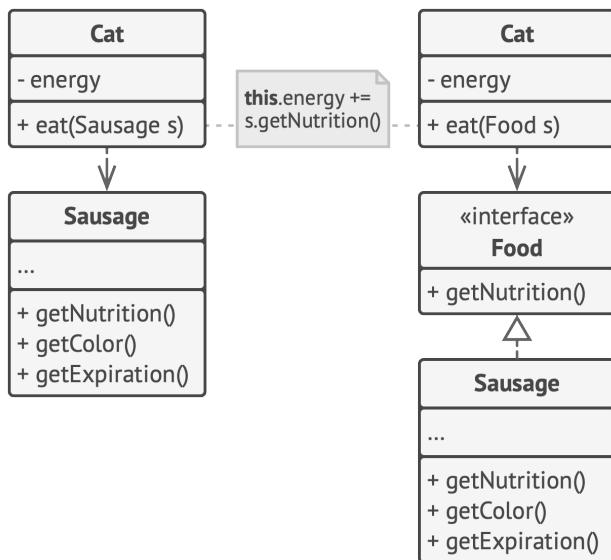
구현이 아닌 인터페이스에 대해 프로그래밍하세요. 또 구상 클래스에 의존하는 대신 추상화에 의존하세요.

기존의 코드를 망가뜨리지 않고 쉽게 확장할 수 있다면 그 디자인은 충분히 유연하다고 말할 수 있습니다. 다른 고양이 예시를 하나 살펴보면서 이 말이 맞는지 확인해 보도록 하죠. 어떤 음식이든 먹을 수 있는 `Cat` (고양이)는 소시지만 먹을 수 있는 고양이보다 유연합니다. 소시지는 '어떤 음식'의 부분 집합이기 때문에 첫 번째 고양이에게 먹일 수 있습니다. 한편 그 고양이의 식사 메뉴는 그 어떤 음식으로도 확장할 수 있죠.

만약 두 클래스가 서로 같이 작업하도록 만들려면 둘 중 하나를 다른 클래스에 의존하게 만드는 것으로 시작할 수 있습니다. 저도 가끔 이 방식으로 시작합니다. 그런데 객체 간의 공동 작업을 만드는 더 유연한 방법이 하나 더 있습니다.

1. 한 객체가 다른 객체에서 정확히 무엇을 필요로 하는지 확인하세요. 어떤 메서드들을 실행하나요?

2. 새 인터페이스 또는 추상 클래스에서 이러한 메서드들을 설명하세요.
3. 다른 객체에 의존하는(dependency인) 클래스가 이 인터페이스를 구현하도록 하세요.
4. 이제 두 번째 클래스를 구상 클래스가 아닌 이 인터페이스에 의존하도록 하세요. 여전히 원래 클래스의 객체들과 작동하도록 만들 수 있지만 이제 연결이 훨씬 더 유연합니다.



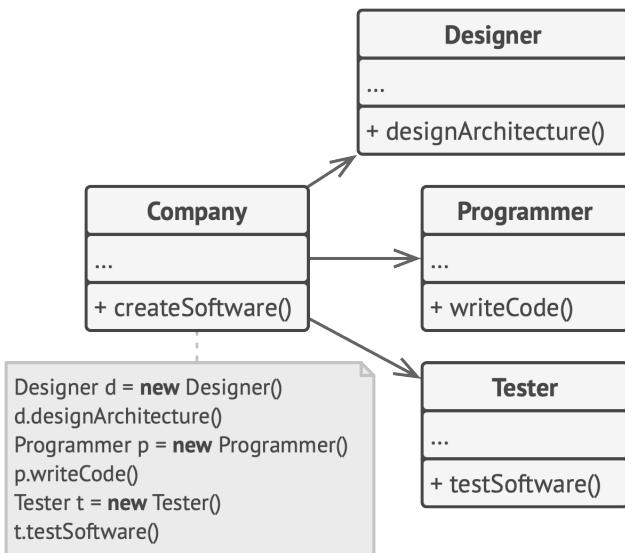
인터페이스를 추출하기 전과 후. 오른쪽 코드는 왼쪽 코드보다 더
유연하지만, 더 복잡하기도 합니다.

이렇게 변경한 뒤 바로 좋아졌다고 느끼지 못할지도 모릅니다.
좋아지긴커녕 코드는 전보다 복잡해졌죠. 하지만 기능을 좀 더
넣을 수 있는 좋은 확장 포인트가 될 수 있겠다고 느껴지거나, 이

코드를 사용하는 다른 사람들이 여기서 뭔가 확장하고 싶어 할 거란 느낌이 든다면 변경을 수행하세요.

예시

이 예시는 객체의 구상 클래스에 의존하는 것보다 그들의 인터페이스를 통해 작업하는 것이 더 유익할 수 있음을 보여줍니다. 당신이 소프트웨어 개발 회사 시뮬레이터를 개발하고 있다고 가정해 봅시다. 그 시뮬레이터에는 여러 직원 유형들을 나타내는 다양한 클래스들이 있을 것입니다.

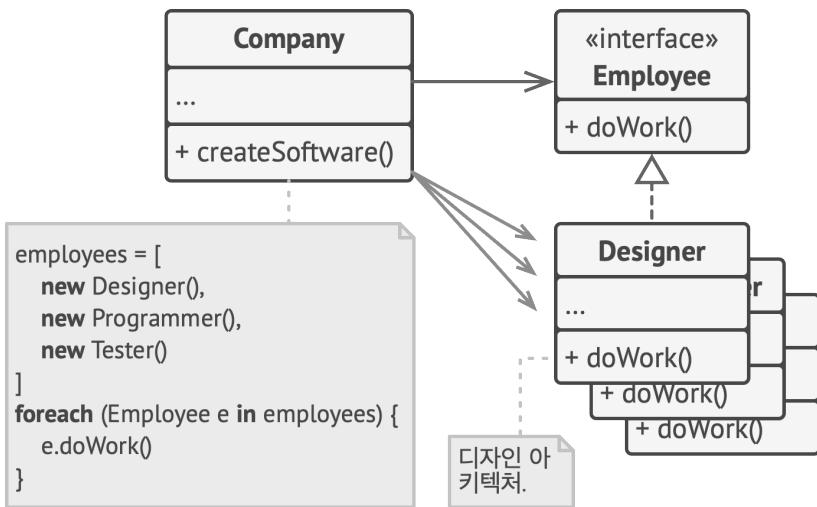


수정 전: 모든 클래스가 단단히 결합하였습니다.

처음에는 **Company** (회사) 클래스가 직원들의 구상 클래스들과 밀접하게 결합되어 있습니다. 한편, 두 클래스의 구현의 차이에도

불구하고 우리는 다양한 업무 관련 메서드들을 일반화한 다음 모든 직원 클래스들에 대한 공통 인터페이스를 추출할 수 있습니다.

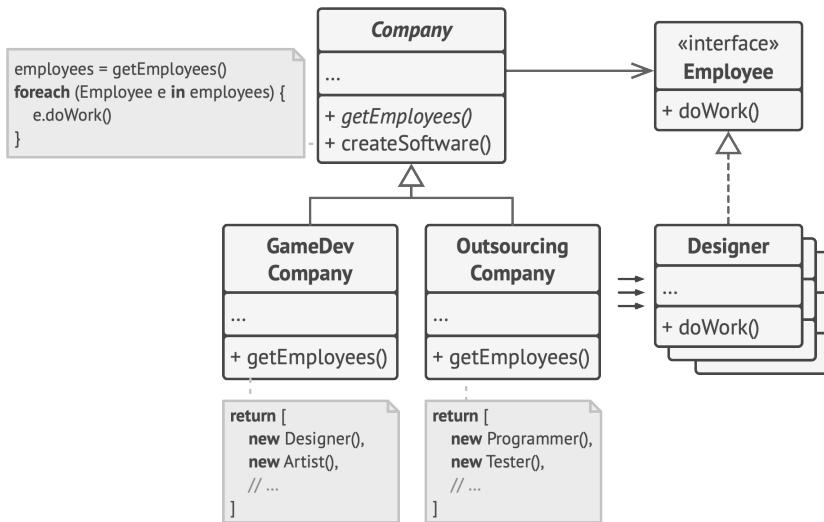
그런 다음 `Company` 클래스 내부에 다형성을 적용하여 `Employee` (직원) 인터페이스를 통해 다양한 직원 객체들을 처리할 수 있습니다.



수정 후: 다형성은 코드를 단순화하는 데 도움이 되었지만 `Company` 클래스의 나머지 부분은 여전히 구상 직원 클래스들에 의존합니다.

`Company` 클래스는 여전히 직원 클래스에 연결되어 있습니다. 이건 좋은 게 아닙니다. 왜냐하면 다른 유형의 직원과 함께 일하는 새로운 유형의 회사들을 소개하려면 `Company` 클래스의 코드를 재사용하는 대신 이 클래스의 대부분을 오버라이드해야 하기 때문입니다.

이 문제를 해결하기 위해 직원들을 가져오는 메서드를 *abstract*로 선언할 수 있습니다. 각 구상 회사는 이 메서드를 다르게 구현할 것이며, 자기 회사에 필요한 직원들만 만들 것입니다.



수정 후: **Company** 클래스의 기본 메서드는 구상 직원 클래스들에 독립적입니다. 직원 객체들은 구상 회사 자식 클래스들에서 생성됩니다.

이렇게 변경한 후 **Company** 클래스는 다양한 직원 클래스에서 독립되었습니다. 이제 기초 회사 클래스의 일부를 계속 재사용하면서 이 클래스를 확장하여 새로운 유형들의 회사들과 직원들을 소개할 수 있습니다. 기초 회사 클래스를 확장해도 이미 그에 의존하고 있는 기존 코드는 손상되지 않습니다.

그건 그렇고, 방금 디자인 패턴이 실제로 어떻게 적용되는지 보셨습니다! 앞에서 나온 내용이 바로 팩토리 메서드 패턴의 예시였습니다. 걱정 마세요. 나중에 이 패턴에 대해 자세히 설명하겠습니다.

상속보다 합성을 사용하세요

클래스들 사이에서 코드를 재사용하는 가장 확실하고 쉬운 방법은 아마도 상속일 겁니다. 예를 들어 같은 코드를 가진 두 개의 클래스가 있다고 가정해 봅시다. 이 두 클래스에 대한 공통 기초 클래스를 만든 후 유사한 코드를 거기로 이동하는 거죠. 누워서 떡 먹기네요!

하지만 상속에는 주의할 점이 있습니다. 이러한 문제들은 프로그램에 이미 엄청나게 많은 클래스가 포함되어 뭔가를 변경하는 것이 상당히 어려워진 뒤에야 확실해지곤 합니다. 다음은 그런 문제들의 목록입니다.

- **자식 클래스는 상위 클래스의 인터페이스를 줄일 수 없습니다.** 자식 클래스는 비록 사용하지 않더라도 부모 클래스의 모든 추상 메서드들을 구현해야 합니다.
- **메서드들을 오버라이드할 때 새 행동이 기초 행동과 호환되는지 확인해야 합니다.** 이것이 중요한 이유는 자식 클래스의 객체들이 부모 클래스의 객체들을 예상하는 모든 코드에 전달될 수 있으며, 코드가 호환되지 않아 충돌하면 안 되기 때문입니다.

- **상속은 부모 클래스의 캡슐화를 깨뜨립니다.** 왜냐하면 부모 클래스 내부의 세부 정보들을 자식 클래스에서 사용할 수 있기 때문입니다. 이와는 반대 상황이 있을 수도 있습니다. 개발자가 추가 확장을 더 쉽게 만들기 위해 상위 클래스가 자식 클래스들의 일부 세부 사항들을 인식하도록 설계할 수도 있겠죠.
- **자식 클래스들은 부모 클래스들과 밀접하게 결합합니다.** 부모 클래스를 변경하면 자식 클래스들의 기능들이 손상될 수 있습니다.
- **상속을 통해 코드를 재사용하려고 하면 병렬 상속 계층구조들이 생성될 수 있습니다.** 상속은 일반적으로 단일 차원에서 발생합니다. 그러나 차원이 두 개 이상이면 그때마다 많은 클래스 조합들을 만들어야 합니다. 그러면 클래스 계층구조는 말도 안 되는 크기로 부풀려질 수 있습니다.

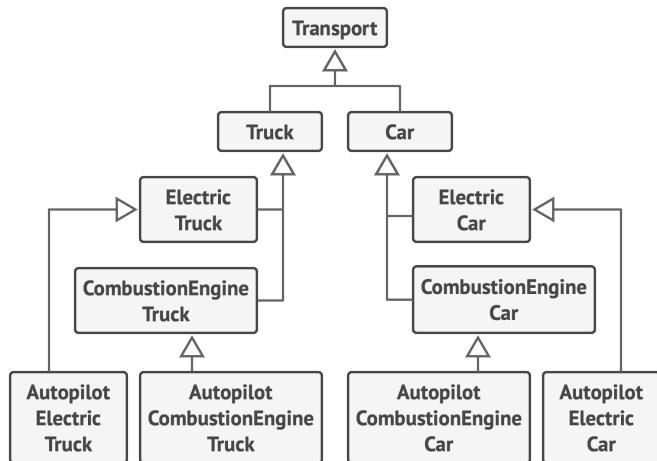
상속에는 합성이라는 대안이 있습니다. 상속은 클래스 간의 '나는 무엇의 ~이다'라는 관계를 나타내지만(예: 자동차는 교통수단이다), 합성은 '~을/를 가진다' 관계들(예: 자동차는 엔진을 가진다)을 나타냅니다.

이 원칙은 집합관계에도 적용될 수 있습니다. 집합관계는 합성의 더욱 완화된 변형으로, 여기서는 한 객체가 다른 객체에 대한 참조를 가질 수는 있지만 이 객체의 수명주기는 관리하지 않습니다. 예를 들어 자동차는 운전자를 가집니다. 그러나

운전자는 차를 사용하지 않거나, 차 없이(가진다의 반대) 그냥 걷을 수도 있습니다.

예시

자동차 제조업체를 위한 카탈로그 앱을 만들어야 한다고 가정해 보세요. 이 회사는 자동차와 트럭을 모두 만듭니다. 차량은 전기 차나 가스 차가 될 수 있습니다. 그리고 모든 모델은 수동 운전 장치나 자동 운전 장치를 가질 수 있습니다.

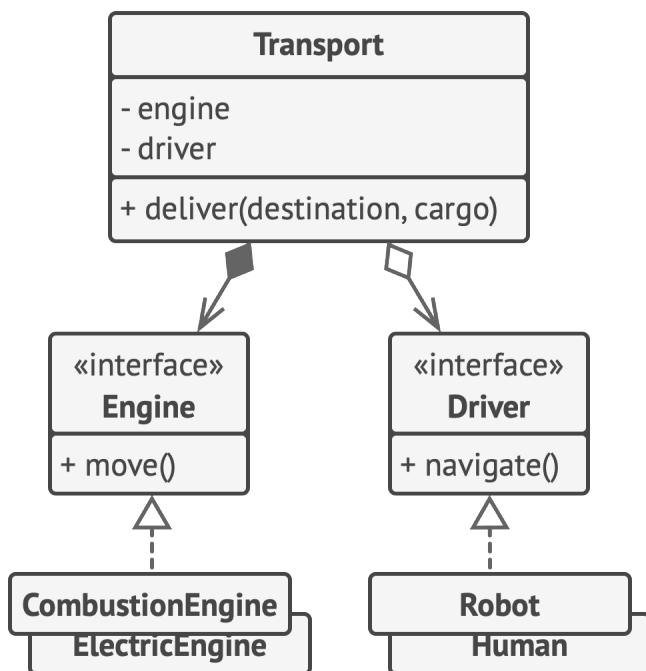


상속: 클래스를 여러 차원(예: 화물 유형 x 엔진 유형 x 내비게이션 유형)으로 확장하면 자식 클래스들의 조합이 폭발적으로 많아질 수 있습니다.

보시다시피 각 추가 매개변수는 자식 클래스들의 수를 증가시킵니다. 또 자식 클래스는 동시에 두 개의 클래스를 확장할 수 있으므로 자식 클래스 간에 중복 코드가 많이 있을 것입니다.

이러한 문제들은 합성으로 해결할 수 있습니다. 자동차 객체들이 자체적으로 행동을 구현하게 하는 대신 다른 객체들에 이를 위임할 수 있습니다.

또 이점이 하나 더 추가되었는데 이는 런타임 때 행동을 바꿀 수 있다는 점입니다. 예를 들어 자동차에 다른 엔진 객체를 할당하여 자동차 객체에 연결된 엔진 객체를 교체할 수 있습니다.



합성: 다양한 기능성의 '차원들'이 각자의 클래스

계층구조들로 추출됩니다.

이 클래스 구조는 이 책의 뒷부분에서 다룰 전략 패턴과 유사합니다.

SOLID 원칙들

기본 디자인 원칙을 배웠으므로 이제 일반적으로 SOLID 원칙들이라고 알려진 5가지 원칙을 살펴봅시다. 로버트 마틴은 자신의 책 『소프트웨어 개발의 지혜: 원칙, 디자인 패턴, 실천방법¹』에서 이 5가지 원칙들을 소개했습니다.

*SOLID*는 소프트웨어 디자인을 보다 이해하기 쉽고 유연하며 유지 관리할 수 있도록 만드는 5가지 설계 원칙들에 대한 연상 기호입니다.

인생의 모든 것이 그렇듯이 이러한 원칙들을 아무 생각 없이 적용하면 득보다 실이 더 많을 수 있습니다. 이러한 원칙들을 프로그램의 아키텍처에 적용하면 프로그램이 필요 이상으로 복잡해질 수 있습니다. 또 이 모든 원칙이 동시에 적용되는 성공적인 소프트웨어 제품이 있는지도 의심스럽습니다. 이러한 원칙들을 적용하기 위해 노력하는 것도 좋지만 항상 실용적이려고 노력하시고 여기에 쓰인 모든 것을 교리로 받아들이지 마세요.

1. 『소프트웨어 개발의 지혜: 원칙, 디자인 패턴, 실천방법』 :
<https://refactoring.guru/ko/principles-book>

S

단일 책임 원칙

ingle Responsibility Principle

클래스는 한 가지 이유로 변경되어야 합니다.

각각의 클래스가 프로그램이 제공하는 기능의 한 부분을 책임지도록 하세요. 그 후 이 책임을 완전히 캡슐화하여 클래스 내부에 숨기세요.

이 원칙의 주목적은 복잡성을 줄이는 것입니다. 코드가 약 200줄에 불과한 프로그램을 위해 정교한 디자인을 만들 필요는 없죠. 십여 개의 메서드들을 멋있게 만들면 그걸로 충분합니다.

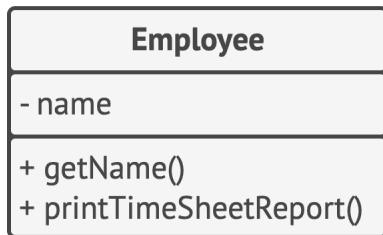
진짜 문제들은 프로그램이 계속해서 성장하고 변경되면서 나타납니다. 어느 시점에 이르면 클래스들은 너무 커져서 더 이상 그 세부 내용을 기억할 수 없게 될 것입니다. 코드 탐색은 매우 느려질 것이고, 클래스나 프로그램의 전체를 훑어봐야 특정 코드를 찾을 수 있게 되겠죠. 프로그램의 너무 많은 인터페이스, 클래스와 유닛의 수가 두통을 유발하면서, 코드에 대한 통제력을 상실하고 있다고 느끼게 될 것입니다.

또 클래스가 너무 많은 작업을 수행하는 경우, 그중 하나가 변경될 때마다 클래스를 변경해야 합니다. 그러면 변경할 생각이 없는 클래스의 다른 부분이 망가질 위험이 있습니다.

프로그램의 특정한 측면에 하나하나 집중하기는 것이 어려워지면 단일 책임 원칙을 떠올려 클래스를 나눠야 하는 것은 아닌지 확인해 보세요.

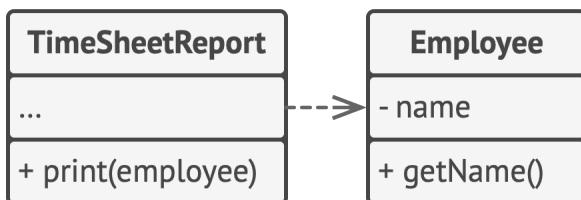
예시

`Employee` (직원) 클래스는 변경되어야 하는 몇 가지 이유가 있습니다. 첫 번째 이유는 클래스의 주 작업인 직원 데이터 관리와 관련이 있을 수 있습니다. 다른 이유도 있습니다. 시간이 흐르면 작업표 보고서의 형식이 변경될 수 있으므로, 그러면 클래스 내에서 코드를 변경해야 합니다.



수정 전: 클래스에는 여러 가지 행동들이 포함되어 있습니다.

작업표 보고서의 인쇄와 관련된 행동을 별도의 클래스로 옮겨 문제를 해결하세요. 이 변경을 통해 다른 보고서 관련 항목들을 새 클래스로 이동할 수 있습니다.



수정 후: 추가 행동들은 자체 클래스들에 있습니다.

O

개방/폐쇄 원칙

Open/Closed Principle

클래스들은 확장에는 열려있어야 하지만 변경에는 닫혀 있어야 합니다.

이 원칙의 주요 목적은 새로운 기능을 구현할 때 기존 코드가 깨지지 않도록 하는 것입니다.

클래스는 확장할 수 있을 때, 자식 클래스를 생성할 수 있을 때, 기초 행동을 오버라이드 하고 새로운 메서드나 필드를 추가하는 등 원하는 모든 작업을 수행할 수 있을 때 *open*(개방)되어 있다고 할 수 있습니다. 일부 프로그래밍 언어들은 `final`과 같은 특수한 키워드를 사용하여 클래스의 추가 확장을 제한할 수 있습니다. 그 후에는 클래스가 더 이상 개방되어 있지 않을 것입니다. 동시에 클래스가 다른 클래스에 의해 사용될 준비가 100% 되어 있다면 해당 클래스는 *closed*(폐쇄) 또는 완료되었다고 할 수 있습니다. 이 때 사용될 준비가 되었다는 것은 인터페이스가 명확하게 정의되어 있으며 미래에 변경되지 않는다는 뜻입니다.

제가 이 원칙에 대해 처음 배웠을 때 저는 혼란스러웠습니다. 왜냐하면 개방과 폐쇄는 서로 공존할 수 없는 조건처럼 들렸기 때문입니다. 그러나 클래스는 확장을 위해 개방됨과 동시에 변경에 대해 폐쇄되어 있을 수 있습니다.

클래스가 이미 개발, 테스트, 검토의 단계를 마쳤고 이미 어떤 프레임워크에 포함되었거나 앱에서 사용되는 경우, 해당 클래스의 코드를 변경하는 것은 위험합니다. 당신은 해당 클래스의 코드를 직접 변경하는 대신 자식 클래스를 만든 후 원래 클래스의 다르게 행동했으면 하는 부분들을 오버라이드할 수 있습니다. 그러면 당신의 목표를 달성하면서도 원래 클래스의 기존 클라이언트를 손상하지 않게 됩니다.

이 원칙은 클래스의 모든 변경 사항에 적용되지 않아도 됩니다. 당신의 클래스에 버그가 있으면 그냥 가서 수정하세요. 그 문제를 수정하려고 자식 클래스를 만들지 말고요. 자식 클래스는 부모 클래스의 문제들에 책임을 져서는 안 됩니다.

예시

당신에게는 모든 배송 메서드들이 내부에 하드코딩되어 있으며 배송 비용을 계산하는 `Order` (주문) 클래스를 갖는 전자 상거래 앱이 있습니다. 새로운 배송 메서드를 추가하려면 `Order` 클래스의 코드를 변경해야 하며, 그러면 코드가 깨질 위험이 있습니다.

Order
- lineItems
- shipping
+ getTotal()
+ getTotalWeight()
+ setShippingType(st)
+ getShippingCost() ○
+ getShippingDate()

```

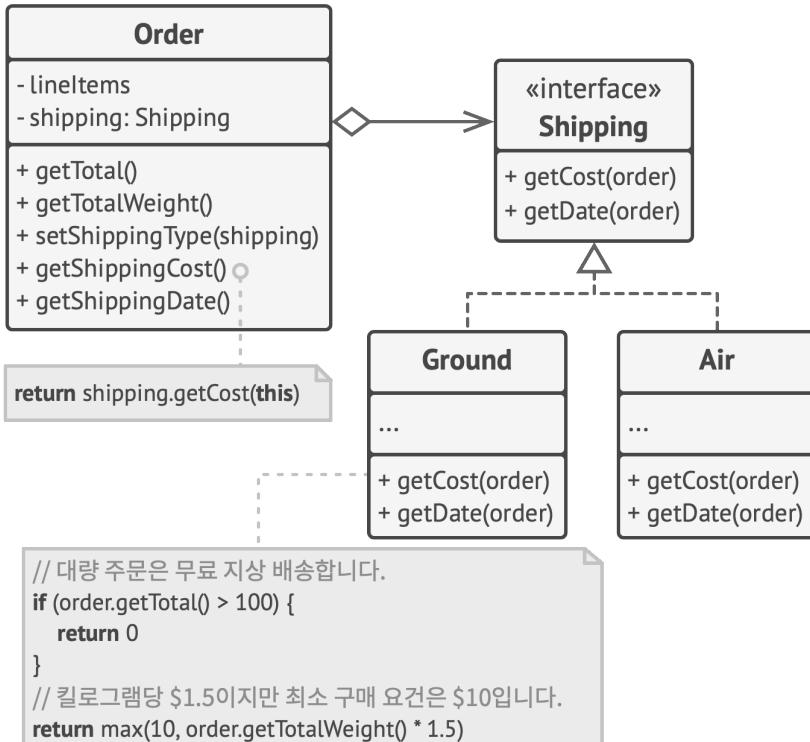
if (shipping == "ground") {
    // 큰 주문은 무료 지상 배송합니다.
    if (getTotal() > 100) {
        return 0
    }
    // 킬로그램당 $1.50이지만 최소 구매
    // 요건은 $10입니다.
    return max(10, getTotalWeight() * 1.5)
}

if (shipping == "air") {
    // 킬로그램당 $3이지만 최소 구매
    // 요건은 $20입니다.
    return max(20, getTotalWeight() * 3)
}

```

수정 전: 앱에 새 배송 메서드를 추가할 때마다 Order 클래스를
변경해야 합니다.

이 문제는 전략 패턴을 적용하여 해결할 수 있습니다. 일단 배송
메서드들을 공통 인터페이스를 가진 별도의 클래스들로
추출하세요.



수정 후: 기존 클래스들을 변경하지 않고 새 배송 메서드를 추가할 수 있습니다.

이제 새로운 배송 메서드를 구현해야 할 때 Order 클래스의 코드를 건드리지 않고 Shipping (배송) 인터페이스에서 새 클래스를 파생할 수 있습니다. Order 클래스의 클라이언트 코드는 사용자가 사용자 인터페이스에서 이 배송 메서드들을 선택할 때마다 주문을 새 클래스의 배송 객체와 연결할 것입니다.

또 이 해결책을 사용하면 단일 책임 원칙에 따라 배달 시간 계산 기능을 보다 관련성 높은 클래스들로 이동할 수 있게 해줍니다.

L

리스코프 치환 원칙

liskov Substitution Principle¹

클래스를 확장할 때 클라이언트 코드를 손상하지 않고 자식 클래스의 객체들을 부모 클래스의 객체들로 교체(치환)할 수 있어야 합니다.

이는 자식 클래스가 부모 클래스의 행동과 계속 호환되어야 함을 의미합니다. 메서드를 오버라이드할 때 기초 행동을 다른 행동들로 완전히 교체하는 대신 확장하세요.

리스코프 치환 원칙은 자식 클래스가 과거에 부모 클래스의 객체들과 함께 작동할 수 있었던 코드와 여전히 호환되는지를 예측하는 데 도움이 되는 일련의 검사들이라고 생각할 수 있습니다. 이 개념은 라이브러리와 프레임워크를 개발할 때 매우 중요합니다. 왜냐하면 당신이 직접 접근하거나 변경할 수 없는 다른 사람들의 코드가 당신의 클래스들을 사용할 것이기 때문입니다.

여러 방식으로 해석이 가능한 다른 디자인 원칙들과는 달리, 리스코프 치환 원칙은 자식 클래스들, 그리고 특히 그들의 메서드들에 대한 일련의 형식적인 요구사항을 갖습니다. 이 요구 사항을 자세히 살펴보려 가봅시다.

-
1. 이 원칙의 이름은 바바라 리스코프가 1987년 자신의 논문 *데이터 추상화와 계층구조*에서 정의했습니다. <https://refactoring.guru/liskov/dah>

- **자식 클래스의 메서드의 매개변수 유형들은 부모 클래스의 메서드의 매개변수 유형들보다 더 추상적이거나 추상화 수준이 같아야 합니다.** 이해가 잘 안 가시나요? 예를 들어보겠습니다.
 - 고양이에게 먹이를 주는 메서드가 있는 `feed(Cat c)`라는 클래스가 있다고 가정해 봅시다. 클라이언트 코드는 항상 `cat` 객체들을 이 메서드에 전달합니다.
 - **좋은 코드:** 위 메서드를 오버라이드한 `feed(Animal c)`라는 자식 클래스를 생성하여 고양이의 부모 클래스인 모든 동물에게 먹이를 줄 수 있도록 했다고 가정해 봅시다. 이제 부모 클래스의 객체 대신 이 자식 클래스의 객체를 클라이언트 코드에 전달해도 모든 것은 여전히 제대로 작동할 것입니다. 이 메서드는 모든 동물에게 먹이를 줄 수 있으므로 클라이언트가 전달하는 모든 고양이에게 먹이를 줄 수 있습니다.
 - **나쁜 코드:** 다른 `feed(BengalCat c)`라는 자식 클래스를 만든 후 해당 클래스의 먹이 주기 메서드를 벵갈 고양이(고양이의 자식 클래스)로만 제한했습니다. 원래 클래스 대신 이와 같은 객체에 클라이언트 코드를 연결하면 어떻게 될까요? 이 메서드는 특정 품종의 고양이에게만 먹이를 줄 수 있으므로 클라이언트가 전달한 일반 고양이에게는 먹이를 제공하지 못하여 모든 관련 기능들이 망가질 것입니다.
- **자식 클래스의 메서드의 반환 유형은 부모 클래스의 메서드의 반환 유형의 하위유형이거나 일치해야 합니다.** 보시다시피, 반환

유형에 대한 요구 사항들은 매개변수 유형에 대한 요구 사항과 정반대입니다.

- `buyCat(): Cat` 메서드가 있는 클래스가 있다고 가정합시다. 클라이언트 코드는 이 메서드를 실행한 결과로 어떤 고양이를 받을 것으로 예상합니다.
- **좋은 코드:** 자식 클래스는 해당 메서드를 다음과 같이 오버라이드합니다. `buyCat(): BengalCat`. 클라이언트는 벵갈 고양이를 받게 되었고, 이건 고양이니까 아무 문제가 없습니다.
- **나쁜 코드:** 자식 클래스는 해당 메서드를 다음과 같이 오버라이드합니다. `buyCat(): Animal`. 이제 클라이언트 코드는 제대로 작동하지 않게 되는데, 그 이유는 고양이를 위해 설계된 구조에 알 수 없는 일반적인 동물(예: 악어? 곰?)을 받기 때문입니다.

또 다른 나쁜 코드의 예시는 동적 타입 프로그래밍 언어에서 찾아볼 수 있습니다. 기초 메서드는 문자열을 반환하는데, 오버라이드된 메서드는 숫자를 반환합니다.

- **자식 클래스의 메서드는 기초 메서드에서 던질 거라 예상되지 않는 예외 유형을 던져서는 안 됩니다.** 다시 말해, 예외 유형들은 기초 메서드가 이미 던질 수 있는 예외 유형들의 하위 유형이거나 그 유형들과 일치하여야 합니다. 이 규칙은 클라이언트 코드의 `try-catch` 블록들이 기초 메서드가 던질 가능성이 있는 특정한 예외

유형들을 대상으로 한다는 사실에서 비롯됩니다. 따라서, 예상치 못한 예외는 클라이언트 코드의 방어선을 통하여 앱 전체를 충돌시킬 수 있습니다.

대부분의 현대 프로그래밍 언어, 특히 정적으로 유형이 지정된 언어(자바, C# 등)에서는 이러한 규칙들이 언어에 내장되어 있어서 해당 규칙들을 위반하는 프로그램은 컴파일할 수 없습니다.

- **자식 클래스는 사전 조건들을 강화해서는 안 됩니다.** 예를 들어 기초 메서드에는 `int` 유형의 매개변수가 있습니다. 자식 클래스가 이 메서드를 오버라이드하고 값이 음수면 예외를 던져서 이 메서드에 전달된 인수의 값이 양수이도록 하면 이는 사전 조건들을 강화합니다. 메서드에 음수들이 전달될 때 잘 작동하던 클라이언트 코드는 이 자식 클래스의 객체와 작업하기 시작하면 망가집니다.
- **자식 클래스는 사후 조건들을 약화해서는 안 됩니다.** 데이터베이스와 함께 작동하는 메서드가 있는 클래스가 있다고 가정해 봅시다. 클래스의 메서드는 값을 반환할 때 항상 열려 있는 모든 데이터베이스 연결을 닫아야 합니다.

당신은 자식 클래스를 생성한 후 데이터베이스 연결을 재사용할 수 있도록 해당 연결을 열린 상태로 유지하도록 자식 클래스를

변경했습니다. 그러나 클라이언트는 당신이 어떤 의도로 그렇게 했는지 전혀 모를 수도 있습니다. 왜냐하면 클라이언트는 메서드들이 모든 연결을 닫을 것으로 예상하기 때문에 메서드를 호출한 직후에 프로그램을 종료하여 시스템을 유령 데이터베이스 연결들로 오염시킬 수 있습니다.

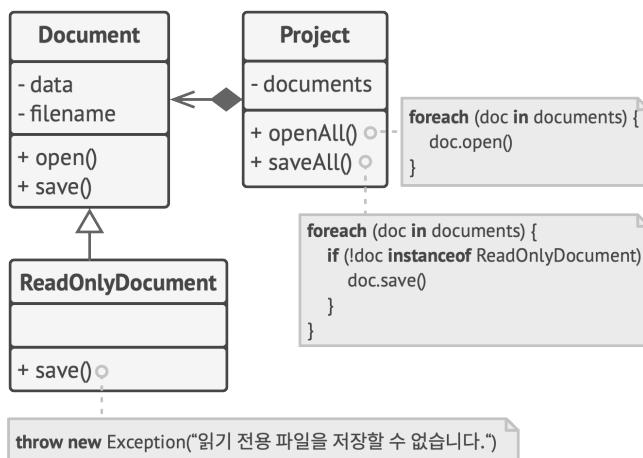
- **부모 클래스의 불변속성들은 반드시 보존되어야 합니다.** 이것은 아마도 리스코프 치환 원칙의 규칙 중 가장 덜 형식적인 규칙일 것입니다. 불변속성들은 비즈니스 요구 사항 및 프로그램 구조에 따라 객체가 해당 객체로 이해되기 위해 갖추어야 하는 조건들입니다. 예를 들어 고양이의 불변속성들은 다리가 4개인 점, 꼬리가 있는 사실, 야옹할 수 있다는 사실 등입니다. 불변속성에 대한 혼란스러운 부분은 그들이 암시적일 수 있다는 사실입니다. 어떤 불변속성들은 인터페이스 계약 또는 메서드 내에 있는 어서션들의 집합의 형태로 명시적으로 선언될 수 있습니다. 그러나 클라이언트 코드의 기대 또는 특정 유닛 테스트들에 의해 암시될 수도 있습니다.

당신이 복잡한 클래스의 모든 불변속성을 이해하거나 인식하지 못할 수 있으므로 불변속성들에 관한 규칙은 위반하기 가장 쉬운 규칙입니다. 따라서 클래스를 확장하는 가장 안전한 방법은 새로운 필드와 메서드를 도입하고 부모 클래스의 기존 멤버들을 변경하지 않는 것입니다. 물론 이 방법이 현실에서 항상 가능한 것은 아닙니다.

- 자식 클래스는 부모 클래스에 있는 비공개 필드의 값을 변경해서는 안 됩니다.** 네? 이게 가능하다고요? 일부 프로그래밍 언어는 반사 메커니즘을 통해 클래스의 비공개 멤버들에 접근할 수 있도록 합니다. 또 다른 언어들(예: 파이썬, 자바스크립트)은 비공개 멤버들을 전혀 보호하지 않습니다.

예시

리스코프 치환 원칙을 위반하는 문서 클래스들의 계층구조에 대한 예시를 한번 살펴봅시다.

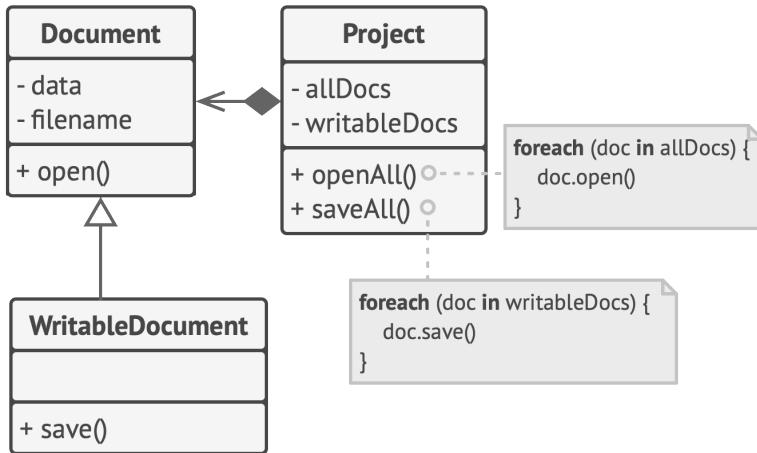


수정 전: 저장하기 기능은 읽기 전용 문서에서는 의미가 없으므로 자식 클래스는 이 문제를 오버라이드된 메서드에서 기초 행동을 재설정하여 해결하려고 합니다.

ReadOnlyDocuments (읽기 전용 문서들) 자식 클래스의 `save` (저장) 메서드는 누군가 호출하려고 하면 예외를 던집니다. 기초

메서드에는 이러한 제한이 없습니다. 이것은 문서를 저장하기 전에 문서의 유형을 확인하지 않으면 클라이언트 코드가 충돌하게 된다는 것을 의미합니다.

또 결과 코드는 개방/폐쇄 원칙을 위반하며, 그 이유는 클라이언트 코드가 문서들의 구상 클래스들에 의존하게 되기 때문입니다. 당신이 새 문서 자식 클래스를 도입하는 경우 이를 지원하도록 클라이언트 코드를 변경해야 합니다.



수정 후: 읽기 전용 문서의 클래스를 계층 구조의 기초 클래스로 만들어
문제가 해결되었습니다.

이 문제는 클래스 계층 구조를 재설계하여 해결할 수 있습니다. 자식 클래스는 부모 클래스의 행동을 확장해야 합니다. 따라서 읽기 전용 문서가 계층구조의 기초 클래스가 됩니다. 쓰기 가능한 문서는 이제 기초 클래스를 확장하고 저장이라는 행동을 추가하는 자식 클래스가 됩니다.

I 인터페이스 분리 원칙

Interface Segregation Principle

클라이언트들은 자신이 사용하지 않는 메서드들에 의존하도록 강요되어서는 안 됩니다.

클라이언트 클래스가 불필요한 행동을 구현할 필요가 없도록 당신의 인터페이스를 적당히 작게 만드세요.

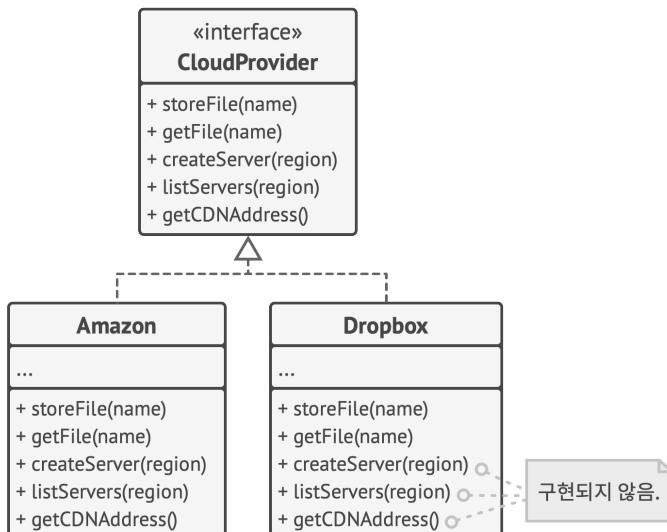
인터페이스 분리 원칙에 따르면, 당신은 '뚱뚱한' 인터페이스를 보다 세분화되고 구체적인 인터페이스들로 나누어야 합니다. 클라이언트들은 자신에게 실제로 필요한 메서드만 구현해야 합니다. 그러지 않고 '뚱뚱한' 인터페이스로 변경한다면 변경된 메서드들을 사용하지 않는 클라이언트들까지 망가뜨리게 됩니다.

클래스 상속은 클래스가 하나의 부모 클래스만 가질 수 있도록 하지만 이 클래스가 동시에 구현할 수 있는 인터페이스들의 수를 제한하지는 않습니다. 따라서 서로 관련 없는 많은 메서드들을 하나의 인터페이스에 다 집어넣을 필요는 없습니다. 이것을 더 정제된 인터페이스들로 나누세요 - 필요하다면 그 전부를 단일 클래스에서 구현할 수도 있습니다. 물론 일부 클래스들은 그중 하나만 구현해도 괜찮을 수 있습니다.

예시

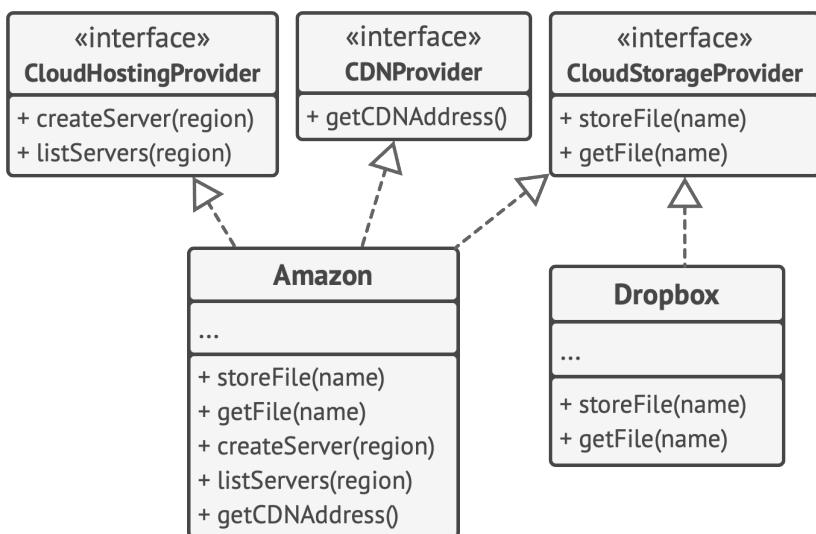
당신이 앱들을 다양한 클라우드 컴퓨팅 공급자들과 쉽게 통합할 수 있는 라이브러리를 개발했다고 가정해 봅시다. 초기 버전에서는 아마존 클라우드만 지원하긴 했지만, 여러 클라우드 서비스와 기능을 전부 다루었습니다.

그 당시에는 모든 클라우드 공급자들이 아마존처럼 광범위한 기능을 가지고 있다고 가정했습니다. 그러나 다른 클라우드 공급자들에 대한 지원을 구현하기 시작했을 때 기존 라이브러리가 가진 대부분의 인터페이스들이 지나치게 광범위하다는 사실을 깨달았습니다. 일부 메서드들은 다른 클라우드 공급자들에게는 없는 기능들을 설명하고 있었죠.



수정 전: 모든 클라이언트가 팽창한 인터페이스의 요구 사항들을 충족하지는 못합니다.

여전히 이러한 메서드들을 구현하고 거기에 일부 스텝들을 넣을 수는 있지만, 그다지 우아한 솔루션은 아닙니다. 더 나은 접근 방식은 인터페이스를 여러 부분으로 나누는 것입니다. 본래의 인터페이스를 구현할 수 있는 클래스들은 이제 몇몇 정제된 인터페이스들만 구현할 수 있습니다. 다른 클래스들은 자신에게 의미가 있는 메서드를 갖는 인터페이스들만 구현할 수 있습니다.



수정 후: 하나의 팽창된 인터페이스가 더 세분된 인터페이스들의 집합으로 나뉩니다

다른 원칙들과 마찬가지로 이 원칙도 너무 많이 사용하게 될 수 있습니다. 인터페이스가 이미 상당히 구체적이라면 더 이상 나누지 마세요. 인터페이스가 많아질수록 코드는 더 복잡해진다는 사실을 잊지 말고 균형을 유지하세요.

D

의존관계 역전 원칙

ependency Inversion Principle

상위 계층 클래스들은 하위 계층 클래스들에 의존해서는 안 됩니다. 둘 다 추상화에 의존해야 합니다. 추상화는 세부 정보들에 의존해서는 안 됩니다. 세부 정보들이 추상화들에 의존해야 합니다.

일반적으로 소프트웨어를 디자인할 때는 클래스를 다음 두 계층으로 분류할 수 있습니다.

- **하위 계층 클래스들은** 디스크와의 작업, 네트워크를 통한 데이터 전송, 데이터베이스 연결 등과 같은 기본 작업을 구현합니다.
- **상위 계층 클래스들은** 하위 계층 클래스들이 무언가를 하도록 지시하는 복잡한 비즈니스 로직을 포함합니다.

때때로 사람들은 하위 계층 클래스들을 먼저 디자인한 다음 상위 계층 클래스들을 디자인하기 시작합니다. 이것은 새 시스템에서 프로토타입을 개발하기 시작할 때 매우 일반적입니다. 왜냐하면 이 시점에서는 하위 계층 기능들이 구현되지 않았거나 명확하지 않기 때문에 상위 계층에서 무엇이 가능한지조차 확신할 수 없기 때문입니다. 이러한 접근 방식을 사용하면 비즈니스 로직 클래스들이 원시(primitive) 하위 계층 클래스들에 의존하게 되는 경향이 있습니다.

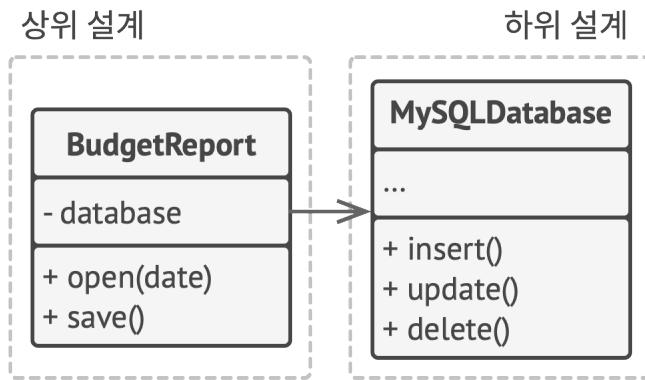
의존관계 역전 원칙은 이러한 의존 관계의 방향을 바꾸자고 제안합니다.

1. 우선, 상위 계층 클래스가 의존하는 하위 계층 작업의 인터페이스를 되도록 비즈니스 용어를 사용해 설명해야 합니다. 예를 들어 비즈니스 로직은 `openFile(x)` (파일 열기), `readBytes(n)` (바이트들 읽기), `closeFile(x)` (파일 닫기) 같은 일련의 메서드를 호출하는 대신 `openReport(file)` (리포트 열기) 메서드를 호출해야 합니다. 이러한 인터페이스들도 상위 계층 인터페이스들로 간주됩니다.
2. 이제 구상 하위 계층 클래스들 대신 이러한 인터페이스에 의존하는 상위 계층 클래스들을 만들 수 있습니다. 이 의존관계는 원래 의존관계보다 훨씬 더 약할 것입니다.
3. 하위 계층 클래스들이 이러한 인터페이스들을 구현하면 이들은 비즈니스 로직 계층에 의존하게 되어 원래 의존관계의 방향이 역전됩니다.

의존관계 역전 원칙은 종종 개방/폐쇄 원칙과 함께 진행됩니다. 당신은 하위 계층 클래스를 확장하여 기존의 클래스들을 손상하지 않고 다른 비즈니스 로직 클래스들과 함께 사용할 수 있습니다.

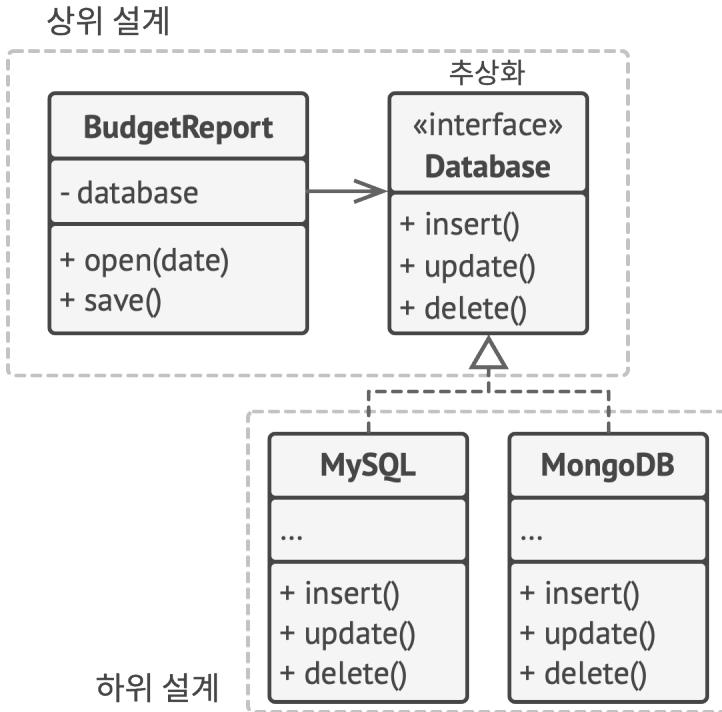
예시

이 예시에서 상위 계층 예산 보고 클래스는 자신의 데이터를 읽고 유지하기 위해 하위 계층 데이터베이스 클래스를 사용합니다. 이것은 하위 계층 클래스에 대한 모든 변경(예: 데이터베이스 서버의 새로운 버전의 출시)이 데이터 저장소의 세부 정보들을 신경 쓰지 않아야 할 상위 계층 클래스에 영향을 미칠 수 있다는 것을 의미합니다.



수정 전: 상위 계층 클래스가 하위 계층 클래스에 의존합니다.

당신은 이 문제를 읽기/쓰기 작업을 설명하는 상위 계층 인터페이스를 생성한 후 예산 보고 클래스가 하위 계층 클래스 대신 이 인터페이스를 사용하도록 하여 해결할 수 있습니다. 그런 다음 원래의 하위 계층 클래스를 비즈니스 로직에 의해 선언된 새로운 읽기/쓰기 인터페이스를 구현하도록 변경하거나 확장할 수 있습니다.



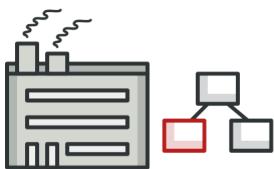
수정 후: 하위 계층 클래스들은 상위 계층 추상화에 의존합니다.

결과적으로 원래 의존 관계의 방향이 역전되었습니다. 이제는 하위 계층의 클래스가 상위 계층의 추상화에 의존합니다.

디자인 패턴 목록

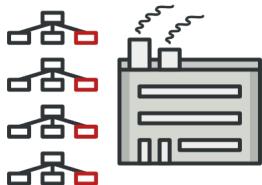
생성 디자인 패턴

생성 디자인 패턴은 기존 코드의 유연성과 재사용을 증가시키는 객체를 생성하는 다양한 방법을 제공합니다.



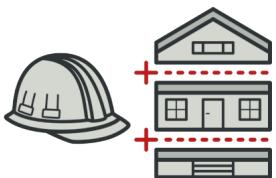
팩토리 메서드

부모 클래스에서 객체를 생성할 수 있는 인터페이스를 제공하지만, 자식 클래스들이 생성될 객체의 유형을 변경할 수 있도록 합니다.



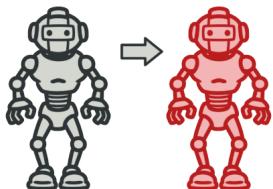
추상 팩토리

관련 객체들의 구상 클래스들을 지정하지 않고도 그들의 패밀리들을 생성할 수 있습니다.



빌더

복잡한 객체들을 단계별로 생성할 수 있도록 합니다. 이 패턴은 같은 생성코드를 사용하여 객체의 다양한 유형들과 표현을 생성할 수 있습니다.



프로토타입

코드를 그들의 클래스들에 의존시키지 않고 기존 객체들을 복사할 수 있도록 합니다.



싱글턴

클래스에 인스턴스가 하나만 있도록 하면서 이 인스턴스에 대한 전역 접근(액세스) 지점을 제공합니다.



팩토리 메서드 패턴

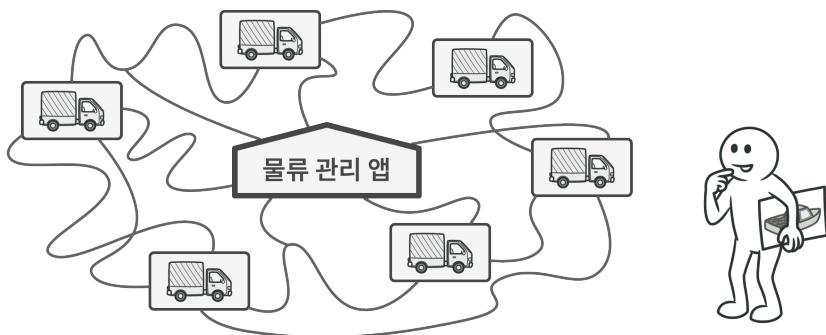
다음 이름으로도 불립니다: 가상 생성자, Factory Method

팩토리 메서드는 부모 클래스에서 객체들을 생성할 수 있는 인터페이스를 제공하지만, 자식 클래스들이 생성될 객체들의 유형을 변경할 수 있도록 하는 생성 패턴입니다.

(:(문제

당신이 물류 관리 앱을 개발하고 있다고 가정합시다. 앱의 첫 번째 버전은 트럭 운송만 처리할 수 있어서 대부분의 코드가 `Truck` (트럭) 클래스에 있습니다.

또 얼마 후 당신의 앱이 유명해졌으며, 매일 해상 물류 회사들로부터 해상 물류 기능을 앱에 추가해 달라는 요청을 수십 개씩 받기 시작했다고 가정해 봅시다.



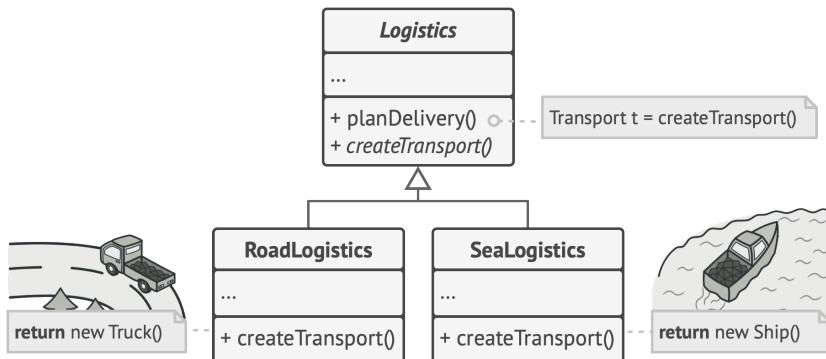
나머지 코드가 이미 기존 클래스들에 결합되어 있다면 프로그램에 새 클래스를 추가하는 일은 그리 간단하지 않습니다.

좋은 소식이죠? 그러나 현재 대부분의 코드는 `Truck` 클래스에 결합되어 있습니다. 앱에 `Ship` (선박) 클래스를 추가하려면 전체 코드 베이스를 변경해야 합니다. 또한 차후 앱에 다른 유형의 교통수단을 추가하려면 아마도 다시 전체 코드 베이스를 변경해야 할 것입니다.

그러면 결과적으로 많은 조건문이 운송 수단 객체들의 클래스에 따라 앱의 행동을 바꾸는 매우 복잡한 코드가 작성될 것입니다.

😊 해결책

팩토리 메서드 패턴은 (`new` 연산자를 사용한) 객체 생성 직접 호출들을 특별한 팩토리 메서드에 대한 호출들로 대체하라고 제안합니다. 걱정하지 마세요: 객체들은 여전히 `new` 연산자를 통해 생성되지만 팩토리 메서드 내에서 호출되고 있습니다. 참고로 팩토리 메서드에서 반환된 객체는 종종 `제품`이라고도 불립니다.

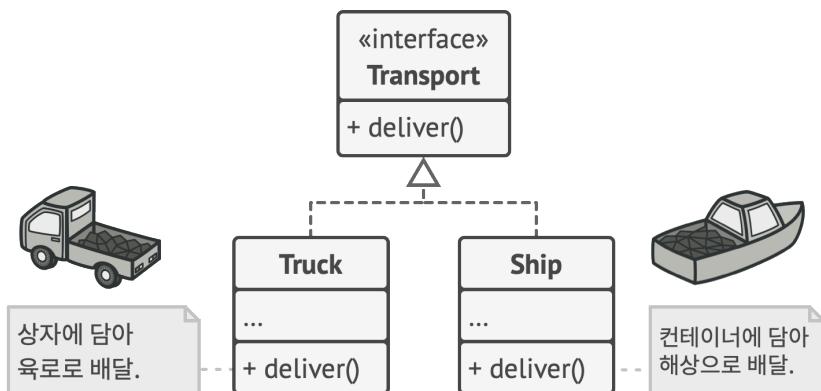


자식 클래스들은 팩토리 메서드가 반환하는 객체들의 클래스를 변경할 수 있습니다.

얼핏 이러한 변경은 무의미해 보일 수도 있는데, 그 이유는 생성자 호출을 프로그램의 한 부분에서 다른 부분으로 옮겼을 뿐이기 때문입니다. 그러나 위와 같은 변경 덕분에 이제 자식 클래스에서

팩토리 메서드를 오버라이딩하고 그 메서드에 의해 생성되는 제품들의 클래스를 변경할 수 있게 되었습니다.

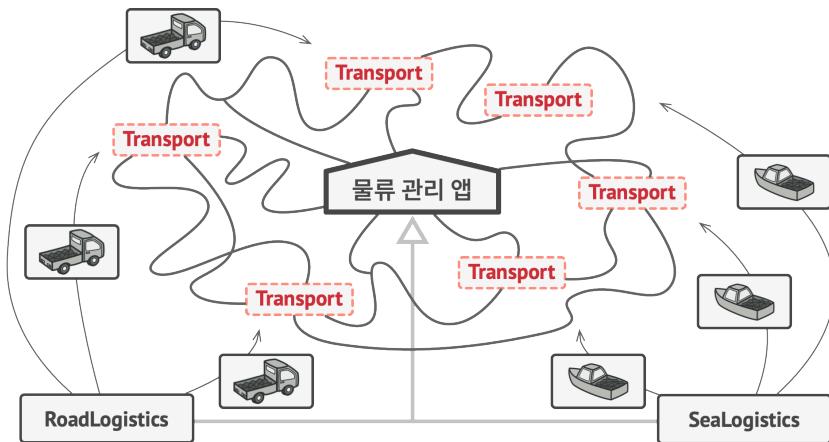
하지만 약간의 제한이 있긴 합니다. 자식 클래스들은 다른 유형의 제품들을 해당 제품들이 공통 기초 클래스 또는 공통 인터페이스가 있는 경우에만 반환할 수 있습니다. 또 이전에 언급한 모든 제품들에 공통인 `Transport` 인터페이스로 `Logistics` 기초 클래스의 `createTransport` 팩토리 메서드의 반환 유형을 선언해야 합니다.



모든 제품들은 같은 인터페이스를 따라야 합니다.

예를 들어 `Truck` 과 `Ship` 클래스들은 모두 `Transport` 인터페이스를 구현해야 하며, 이 인터페이스는 `deliver` (배달)라는 메서드를 선언합니다. 그러나 각 클래스는 이 메서드를 다르게 구현합니다. 트럭은 육로로 화물을 배달하고 선박은 해상으로 화물을 배달합니다. `RoadLogistics` (도로 물류)

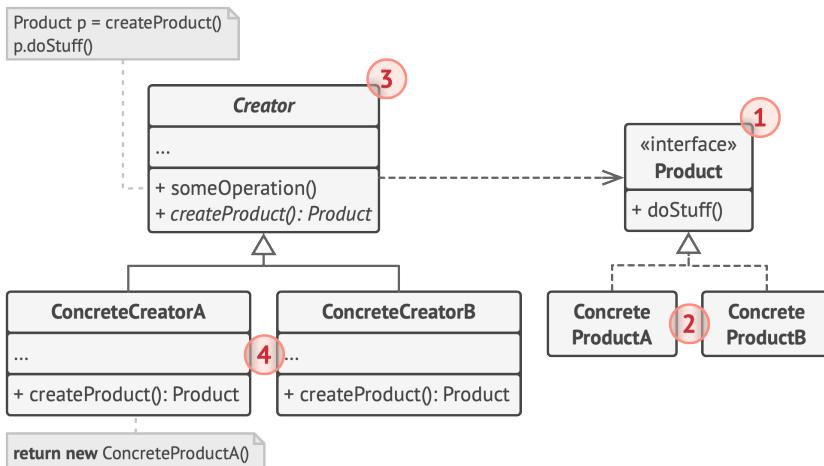
클래스에 포함된 팩토리 메서드는 `Truck` 객체들을 반환하는 반면 `SeaLogistics` (해운 물류) 클래스에 포함된 팩토리 메서드는 선박 객체들을 반환합니다.



모든 제품 클래스들이 공통 인터페이스를 구현하는 한, 제품 클래스들의 객체들을 손상하지 않고 클라이언트 코드를 통과시킬 수 있습니다.

팩토리 메서드를 사용하는 코드를 종종 `클라이언트` 코드라고 부르며, 클라이언트 코드는 다양한 자식 클래스들에서 실제로 반환되는 여러 제품 간의 차이에 대해 알지 못합니다. 클라이언트 코드는 모든 제품을 추상 `Transport` (운송체계)로 간주합니다. 클라이언트는 모든 `Transport` 객체들이 `deliver` (배달) 메서드를 가져야 한다는 사실을 잘 알고 있지만, 이 메서드가 정확히 어떻게 작동하는지는 클라이언트에게 중요하지 않습니다.

구조



- 제품은 인터페이스를 선언합니다. 인터페이스는 생성자와 자식 클래스들이 생성할 수 있는 모든 객체에 공통입니다.
- 구상 제품들은 제품 인터페이스의 다양한 구현들입니다.
- 크리에이터(Creator) 클래스는 새로운 제품 객체들을 반환하는 팩토리 메서드를 선언합니다. 중요한 점은 이 팩토리 메서드의 반환 유형이 제품 인터페이스와 일치해야 한다는 것입니다.

당신은 팩토리 메서드를 `abstract` (추상)로 선언하여 모든 자식 클래스들이 각각 이 메서드의 자체 버전들을 구현하도록 강제할 수 있으며, 또 대안적으로 기초 팩토리 메서드가 디플트(기본값) 제품 유형을 반환하도록 만들 수도 있습니다.

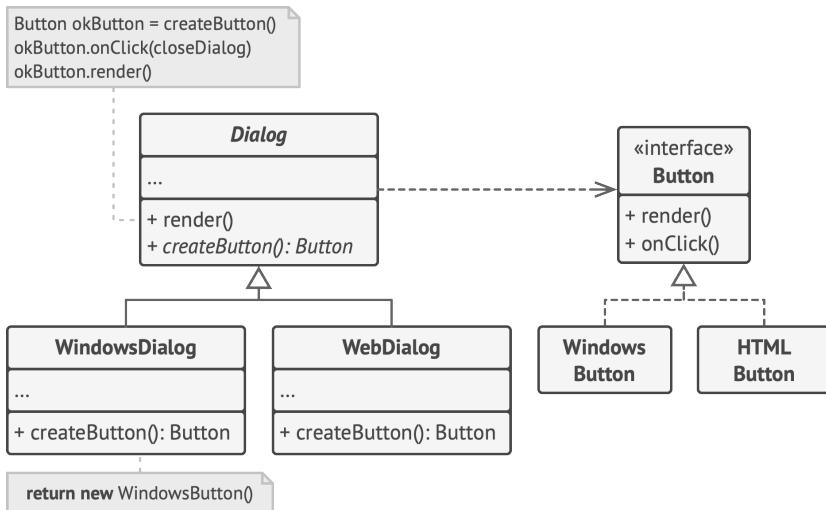
크리에이터라는 이름에도 불구하고 크리에이터의 주책임은 제품을 생성하는 것이 **아닙니다.** 일반적으로 크리에이터 클래스에는 이미 제품과 관련된 핵심 비즈니스 로직이 있으며, 팩토리 메서드는 이 로직을 구상 제품 클래스들로부터 디커플링(분리) 하는 데 도움을 줄 뿐입니다. 대규모 소프트웨어 개발사에 비유해 보자면, 이 회사는 프로그래머들을 위한 교육 부서가 있을 수 있으나, 회사의 주 임무는 프로그래머를 교육하는 것이 아니라 코드를 작성하는 것입니다.

4. **구상 크리에이터**들은 기초 팩토리 메서드를 오버라이드(재정의)하여 다른 유형의 제품을 반환하게 하도록 합니다.

참고로 팩토리 메서드는 항상 새로운 인스턴스들을 **생성해야** 할 필요가 없습니다. 팩토리 메서드는 기존 객체들을 캐시, 객체 풀 또는 다른 소스로부터 반환할 수 있습니다.

의사코드

아래 예시는 어떻게 **팩토리 메서드**가 클라이언트 코드를 구상 UI 클래스들과 결합하지 않고도 크로스 플랫폼 UI 요소들을 생성할 수 있는지를 보여줍니다.



크로스 플랫폼 대화상자 예시.

기초 `Dialog` (대화상자) 클래스는 여러 UI 요소들을 사용하여 대화 상자를 렌더링합니다. 다양한 운영 체제에서 이러한 요소들은 약간씩 다르게 보일 수 있지만 여전히 일관되게 작동해야 합니다. 예를 들어 윈도우에서의 버튼은 리눅스에서도 여전히 버튼이어야 합니다.

팩토리 메서드가 적용되면, 당신은 대화상자 로직을 각 운영 체제를 위하여 반복해서 재작성할 필요가 없습니다. 기초 `Dialog` 클래스 내에서 버튼을 생성하는 팩토리 메서드를 선언하면 나중에 팩토리 메서드에서 윈도우 유형의 버튼들을 반환하는 `Dialog` 자식 클래스를 생성할 수 있습니다. 그 후 이 자식 클래스는 기초 클래스로부터 `Dialog`의 코드 대부분을 상속받으나, 팩토리 메서드 덕분에 윈도우 유형의 버튼들도 렌더링할 수 있습니다.

이 패턴이 작동하려면 기초 Dialog 클래스가 추상 버튼들과 함께 작동해야 합니다. (참고로 추상 버튼은 모든 구상 버튼들이 따르는 인터페이스 또는 기초 클래스입니다). 이렇게 해야 다이얼로그 코드가 버튼 유형에 관계없이 작동합니다.

물론, 위 접근 방식을 다른 UI 요소들에도 적용할 수 있으나, 대화 상자에 새로운 팩토리 메서드를 추가할 때마다 이 프로그램은 **추상 팩토리** 패턴에 더 가까워집니다. 걱정하지 마세요. 추상 팩토리 패턴에 대해서는 나중에 설명하겠습니다.

```

1 // 크리에이터 클래스는 제품 클래스의 객체를 반환해야 하는 팩토리 메서드를
2 // 선언합니다. 크리에이터의 자식 클래스들은 일반적으로 이 메서드의 구현을
3 // 제공합니다.
4 class Dialog is
5     // 크리에이터는 팩토리 메서드의 일부 디폴트 구현을 제공할 수도 있습니다.
6     abstract method createButton():Button
7
8     // 크리에이터의 주 업무는 제품을 생성하는 것이 아닙니다. 크리에이터는
9     // 일반적으로 팩토리 메서드에서 반환된 제품 객체에 의존하는 어떤 핵심
10    // 비즈니스 로직을 포함합니다. 자식 클래스들은 팩토리 메서드를 오버라이드 한
11    // 후 해당 메서드에서 다른 유형의 제품을 반환하여 해당 비즈니스 로직을
12    // 간접적으로 변경할 수 있습니다.
13    method render() is
14        // 팩토리 메서드를 호출하여 제품 객체를 생성하세요.
15        Button okButton = createButton()
16        // 이제 제품을 사용하세요.
17        okButton.onClick(closeDialog)
18        okButton.render()
19

```

```
20
21 // 구상 크리에이터들은 결과 제품들의 유형을 변경하기 위해 팩토리 메서드를
22 // 오버라이드합니다.
23 class WindowsDialog extends Dialog is
24     method createButton():Button is
25         return new WindowsButton()
26
27 class WebDialog extends Dialog is
28     method createButton():Button is
29         return new HTMLButton()
30
31
32 // 제품 인터페이스는 모든 구상 제품들이 구현해야 하는 작업들을 선언합니다.
33 interface Button is
34     method render()
35     method onClick(f)
36
37 // 구상 제품들은 제품 인터페이스의 다양한 구현을 제공합니다.
38 class WindowsButton implements Button is
39     method render(a, b) is
40         // 버튼을 윈도우 스타일로 렌더링하세요.
41     method onClick(f) is
42         // 네이티브 운영체제 클릭 이벤트를 바인딩하세요.
43
44 class HTMLButton implements Button is
45     method render(a, b) is
46         // 버튼의 HTML 표현을 반환하세요.
47     method onClick(f) is
48         // 웹 브라우저 클릭 이벤트를 바인딩하세요.
49
50
51 class Application is
```

```

52   field dialog: Dialog
53
54   // 앱은 현재 설정 또는 환경 설정에 따라 크리에이터의 유형을 선택합니다.
55   method initialize() is
56     config = readApplicationConfigFile()
57
58     if (config.OS == "Windows") then
59       dialog = new WindowsDialog()
60     else if (config.OS == "Web") then
61       dialog = new WebDialog()
62     else
63       throw new Exception("Error! Unknown operating system.")
64
65   // 클라이언트 코드는 비록 구상 크리에이터의 기초 인터페이스를 통하는 것이긴
66   // 하지만 구상 크리에이터의 인스턴스와 함께 작동합니다. 클라이언트가
67   // 크리에이터의 기초 인터페이스를 통해 크리에이터와 계속 작업하는 한 모든
68   // 크리에이터의 자식 클래스를 클라이언트에 전달할 수 있습니다.
69   method main() is
70     this.initialize()
71     dialog.render()

```

💡 적용

💡 팩토리 메서드는 당신의 코드가 함께 작동해야 하는 객체들의 정확한 유형들과 의존관계들을 미리 모르는 경우 사용하세요.

⚡ 팩토리 메서드는 제품 생성 코드를 제품을 실제로 사용하는 코드와 분리합니다. 그러면 제품 생성자 코드를 나머지 코드와는 독립적으로 확장하기 쉬워집니다.

예를 들어, 앱에 새로운 제품을 추가하려면 당신은 새로운 크리에이터 자식 클래스를 생성한 후 해당 클래스 내부의 팩토리 메서드를 오버라이딩(재정의)하기만 하면 됩니다.

 팩토리 메서드는 당신의 라이브러리 또는 프레임워크의 사용자들에게 내부 컴포넌트들을 확장하는 방법을 제공하고 싶을 때 사용하세요.

 상속(inheritance)은 아마도 라이브러리나 프레임워크의 디폴트 행동을 확장하는 가장 쉬운 방법일 것입니다. 그러나 프레임워크는 표준 컴포넌트 대신 당신의 자식 클래스를 사용해야 한다는 것을 어떻게 인식할까요?

해결책은 일단 프레임워크 전체에서 컴포넌트들을 생성하는 코드를 단일 팩토리 메서드로 줄인 후, 누구나 이 팩토리 메서드를 오버라이드 할 수 있도록 하는 것입니다.

그러면 해결책의 예시를 한번 살펴봅시다. 당신이 오픈 소스 UI 프레임워크를 사용하여 앱을 작성하고 있고, 당신이 개발 중인 앱에는 등근 버튼들이 필요한데 프레임워크는 사각형 버튼만 제공한다고 가정합시다. 또 표준 `Button`(버튼) 클래스는 `RoundButton`(등근 버튼) 자식 클래스로 확장했지만, 이제 메인 `UIFramework`(사용자 인터페이스 프레임워크) 클래스에 디폴트 클래스 대신 새로운 `RoundButton`(등근 버튼) 자식 클래스를 사용하라고 지시해야 한다고 가정해 봅시다.

이를 달성하려면 기초 프레임워크 클래스에서 자식 클래스 `UIWithRoundButtons` 를 만들어서 기초 프레임워크 클래스의 `createButton` 메서드를 오버라이딩(재정의)합니다. 이 메서드는 기초 클래스에 `Button` 객체들을 반환하지만, 당신은 당신의 자식 클래스가 `RoundButton` 객체들을 반환하도록 만듭니다. 이제 `UIFramework` 클래스 대신 `UIWithRoundButtons` 클래스를 사용하면 끝입니다!

☞ 팩토리 메서드는 기존 객체들을 매번 재구축하는 대신 이들을 재사용하여 시스템 리소스를 절약하고 싶을 때 사용하세요.

☞ 이러한 요구 사항은 데이터베이스 연결, 파일 시스템 및 네트워크처럼 시스템 자원을 많이 사용하는 대규모 객체들을 처리할 때 자주 발생합니다.

기존 객체를 재사용하려면 무엇을 해야 하는지 한번 생각해 봅시다.

1. 먼저 생성된 모든 객체를 추적하기 위해 일부 스토리지를 생성해야 합니다.
2. 누군가가 객체를 요청하면 프로그램은 해당 풀 내에서 유휴(free) 객체를 찾아야 합니다. 그 후...
3. ...이 객체를 클라이언트 코드에 반환해야 합니다.

4. 유휴(free) 객체가 없으면, 프로그램은 새로운 객체를 생성해야 합니다. (그리고 풀에 이 객체를 추가해야 합니다).

이것은 정말로 많은 양의 코드입니다! 그리고 프로그램을 중복 코드로 오염시키지 않도록 이 많은 양의 코드를 모두 한곳에 넣어야 합니다.

아마도 이 코드를 배치할 수 있는 가장 확실하고 편리한 위치는 우리가 재사용하려는 객체들의 클래스의 생성자일 것입니다. 그러나 생성자는 특성상 항상 **새로운 객체들을** 반환해야 하며, 기존 인스턴트를 반환할 수는 없습니다.

따라서 새 객체들을 생성하고 기존 객체를 재사용할 수 있는 일반적인 메서드가 필요합니다. 이 설명, 꼭 팩토리 메서드처럼 들리지 않나요?

구현방법

1. 모든 제품이 같은 인터페이스를 따르도록 하세요. 이 인터페이스는 모든 제품에서 의미가 있는 메서드들을 선언해야 합니다.
2. 크리에이터 클래스 내부에 빈 팩토리 메서드를 추가하세요. 이 메서드의 반환 유형은 공통 제품 인터페이스와 일치해야 합니다.

3. 크리에이터의 코드에서 제품 생성자들에 대한 모든 참조를 찾으세요. 이 참조들을 하나씩 팩토리 메소드에 대한 호출로 교체하면서 제품 생성 코드를 팩토리 메서드로 추출하세요.

반환된 제품의 유형을 제어하기 위해 팩토리 메서드에 임시 매개변수를 추가해야 할 수도 있습니다.

이 시점에서 팩토리 메서드의 코드는 꽤 복잡할 수 있습니다. 예를 들어 인스턴트화할 제품 클래스를 선택하는 큰 `switch` 문장이 있을 수 있습니다. 하지만 걱정하지 마세요, 곧 이 문제를 해결할 테니까요.

4. 이제 팩토리 메서드에 나열된 각 제품 유형에 대한 크리에이터 자식 클래스들의 집합을 생성한 후, 자식 클래스들에서 팩토리 메서드를 오버라이딩하고 기초 메서드에서 생성자 코드의 적절한 부분들을 추출하세요.
5. 제품 유형이 너무 많아 모든 제품에 대하여 자식 클래스들을 만드는 것이 합리적이지 않을 경우, 자식 클래스들의 기초 클래스의 제어 매개변수를 재사용할 수 있습니다.

예를 들어, 다음과 같은 클래스 계층구조가 있다고 상상해 보세요. `Mail` (우편) 기초 클래스의 자식 클래스들은 `AirMail` (항공우편) 과 `GroundMail` (지상우편)이며, `Transport` (운송수단) 클래스의 자식 클래스들은 `Plane` (비행기), `Truck` (트럭), 그리고 `Train` (기차)입니다. `AirMail` (항공우편) 클래스는

`Plane` (비행기) 객체만 사용하지만, `GroundMail` (지상우편)은 `Truck` 과 `Train` 객체들 모두 사용할 수 있습니다. 이 두 가지 경우를 모두 처리하기 위해 새 자식 클래스(예: `TrainMail` (기차우편))를 만들 수도 있으나, 다른 방법도 있습니다. 클라이언트 코드가 받으려는 제품을 제어하기 위해 `GroundMail` 클래스의 팩토리 메서드에 전달인자(argument)를 전달하는 방법입니다.

6. 추출이 모두 끝난 후 기초 팩토리 메서드가 비어 있으면, 해당 팩토리 메서드를 추상화할 수 있습니다. 팩토리 메서드가 비어 있지 않으면, 나머지를 그 메서드의 디폴트 행동으로 만들 수 있습니다.

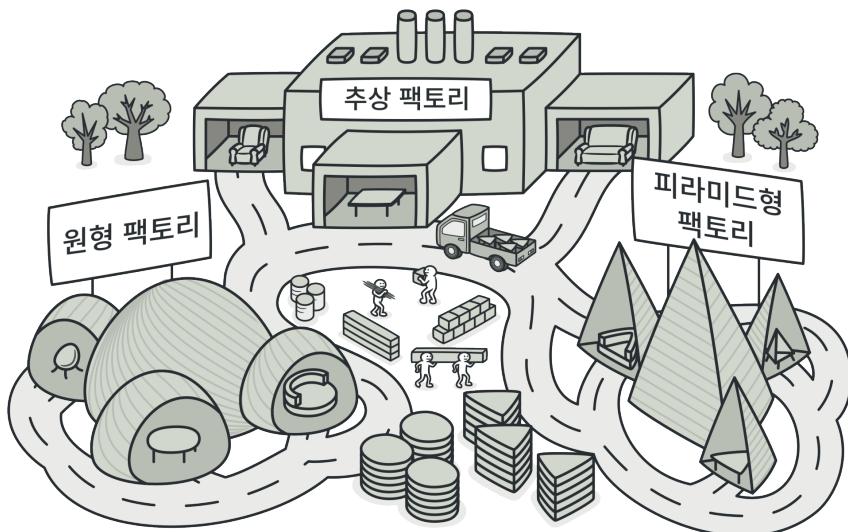
⚠️ 장단점

- ✓ 크리에이터와 구상 제품들이 단단하게 결합되지 않도록 할 수 있습니다.
- ✓ 단일 책임 원칙. 제품 생성 코드를 프로그램의 한 위치로 이동하여 코드를 더 쉽게 유지관리할 수 있습니다.
- ✓ 개방/폐쇄 원칙. 당신은 기존 클라이언트 코드를 훼손하지 않고 새로운 유형의 제품들을 프로그램에 도입할 수 있습니다.
- ✗ 패턴을 구현하기 위해 많은 새로운 자식 클래스들을 도입해야 하므로 코드가 더 복잡해질 수 있습니다. 가장 좋은 방법은

크리에이터 클래스들의 기존 계층구조에 패턴을 도입하는 것입니다.

↔ 다른 패턴과의 관계

- 많은 디자인은 복잡성이 낮고 자식 클래스들을 통해 더 많은 커스터마이징이 가능한 팩토리 메서드로 시작해 더 유연하면서도 더 복잡한 추상 팩토리, 프로토타입 또는 빌더 패턴으로 발전해 나갑니다.
- 추상 팩토리 클래스들은 팩토리 메서드들의 집합을 기반으로 하는 경우가 많습니다. 그러나 당신은 또한 프로토타입을 사용하여 추상 팩토리의 구상 클래스들의 생성 메서드들을 구현할 수도 있습니다.
- 팩토리 메서드를 반복자와 함께 사용하여 컬렉션 자식 클래스들이 해당 컬렉션들과 호환되는 다양한 유형의 반복자들을 반환하도록 할 수 있습니다.
- 프로토타입은 상속을 기반으로 하지 않으므로 상속과 관련된 단점들이 없습니다. 반면에 프로토타입은 복제된 객체의 복잡한 초기화가 필요합니다. 팩토리 메서드는 상속을 기반으로 하지만 초기화 단계가 필요하지 않습니다.
- 팩토리 메서드는 템플릿 메서드의 특수화라고 생각할 수 있습니다. 동시에 대규모 템플릿 메서드의 한 단계의 역할을 팩토리 메서드가 할 수 있습니다.



추상 팩토리 패턴

다음 이름으로도 불립니다: Abstract Factory

추상 팩토리는 관련 객체들의 구상 클래스들을 지정하지 않고도 관련 객체들의 모음을 생성할 수 있도록 하는 생성패턴입니다.

(:) 문제

예를 들어 당신이 가구 판매장을 위한 프로그램을 만들고 있다고 가정합시다. 당신의 코드는 다음을 나타내는 클래스들로 구성됩니다:

- 관련 제품들로 형성된 패밀리(제품군), 예: Chair (의자) + Sofa (소파) + CoffeeTable (커피 테이블).
- 해당 제품군의 여러 가지 변형. 예를 들어 Chair (의자) + Sofa (소파) + CoffeeTable (커피 테이블) 같은 제품들은 Modern (현대식), Victorian (빅토리안), ArtDeco (아르데코 양식)와 같은 변형으로 제공됩니다.



제품 패밀리들과 그들의 변형들

이제 당신이 새로운 개별 가구 객체를 생성했을 때, 이 객체들이 기존의 같은 패밀리 내에 있는 다른 가구 객체들과 일치하는 변형(스타일)을 가지도록 할 방법이 필요합니다. 그 이유는 당신의 고객이 스타일이 일치하지 않는 가구 세트를 받으면 크게 실망할 수 있기 때문입니다.



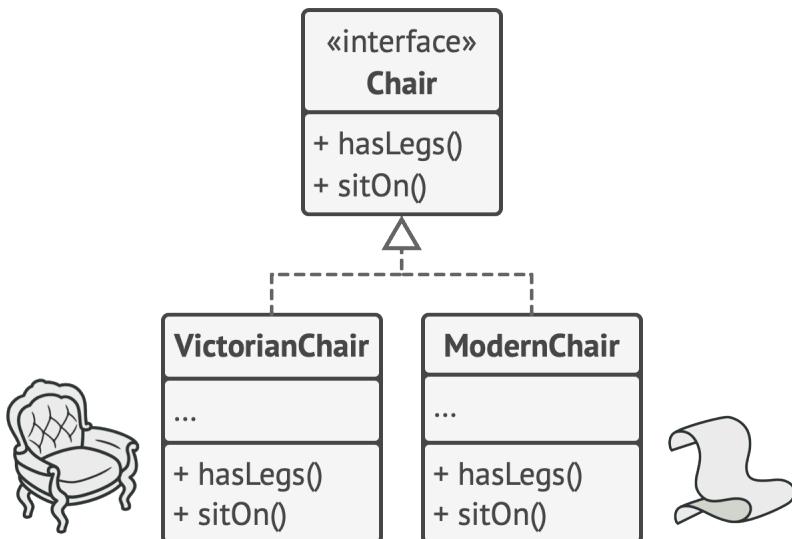
현대식 스타일의 소파는 빅토리안 스타일의 의자들과 잘 어울리지 않습니다.

또, 가구 공급업체들은 카탈로그를 매우 자주 변경하기 때문에, 그들은 새로운 제품 또는 제품군(패밀리)을 추가할 때마다 기존 코드를 변경해야 하는 번거로움을 피하고 싶을 것입니다.

😊 해결책

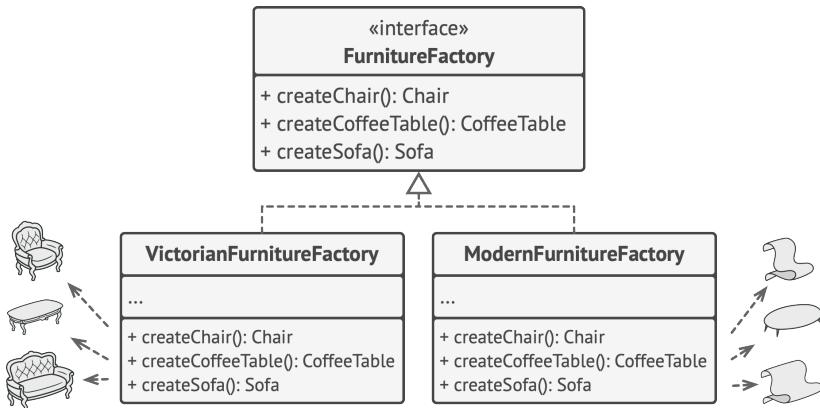
추상 공장 패턴의 첫 번째 방안은 각 제품 패밀리(제품군)에 해당하는 개별적인 인터페이스를 명시적으로 선언하는 것입니다. (예: 의자, 소파 또는 커피 테이블). 그다음, 제품의 모든 변형이 위

인터페이스를 따르도록 합니다. 예를 들어, 모든 의자의 변형들은 Chair (의자) 인터페이스를 구현한다; 모든 커피 테이블 변형들은 CoffeeTable (커피 테이블) 인터페이스를 구현한다, 등의 규칙을 명시합니다.



같은 객체의 모든 변형은 단일 클래스 계층구조로 옮겨져야 합니다.

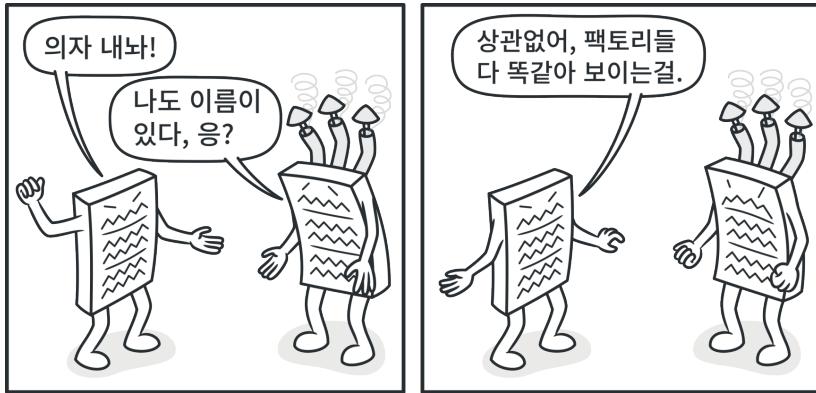
추상 공장 패턴의 다음 단계는 추상 팩토리 패턴을 선언하는 것입니다. 추상 공장 패턴은 제품 패밀리 내의 모든 개별 제품들의 생성 메서드들이 목록화되어있는 인터페이스입니다. (예: `createChair` (의자 생성), `createSofa` (소파 생성), `createCoffeeTable` (커피 테이블 생성) 등).



각 구상 팩토리는 특정 제품의 변형에 해당합니다.

다음은 제품 변형을 다룰 차례입니다. 제품 패밀리의 각 변형에 대해 `AbstractFactory` 추상 팩토리 인터페이스를 기반으로 별도의 팩토리 클래스를 생성합니다. 팩토리는 특정 종류의 제품을 반환하는 클래스입니다. 예를 들어 `ModernFurnitureFactory` (현대식 가구 팩토리)에서는 다음 객체들만 생성할 수 있습니다: `ModernChair` (현대식 의자), `ModernSofa` (현대식 소파) 및 `ModernCoffeeTable` (현대식 커피 테이블).

클라이언트 코드는 자신에 해당하는 추상 인터페이스를 통해 팩토리들과 제품들 모두와 함께 작동해야 합니다. 그래야 클라이언트 코드에 넘기는 팩토리의 종류와 제품 변형들을 클라이언트 코드를 손상하지 않으며 자유자재로 변경할 수 있습니다.

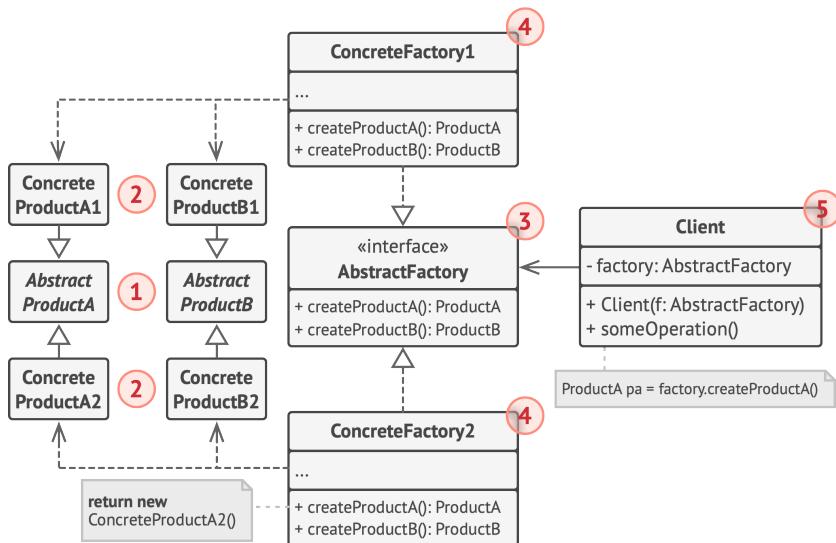


클라이언트들은 함께 작업하는 팩토리의 구상 클래스에 대해 신경을 쓰지 않아도 돼야 합니다.

클라이언트가 팩토리에 의자를 주문했다고 가정해 봅시다. 클라이언트는 팩토리의 클래스들을 알 필요가 없으며, 팩토리가 어떤 변형의 의자를 생성할지 전혀 신경을 쓰지 않습니다. 클라이언트는 추상 `Chair` (의자) 인터페이스를 사용하여, 현대식 의자이든 빅토리아식 의자이든 종류에 상관없이 모든 의자를 항상 동일한 방식으로 주문하며, 그가 의자에 대해 아는 유일한 사실은 제품이 `sitOn` (앉을 수 있다) 메서드를 구현한다는 것뿐입니다. 그러나, 생성된 의자의 변형은 항상 같은 팩토리 객체에서 생성된 소파 또는 커피 테이블의 변형과 같을 것입니다.

여기에서 명확히 짚고 넘어가야 할 점이 있습니다. 클라이언트가 추상 인터페이스에만 노출된다면 실제 팩토리 객체를 생성하는 것은 무엇일까요? 일반적으로 프로그램은 초기화 단계에서 구상 팩토리 객체를 생성합니다. 그 직전에 프로그램은 환경 또는 구성 설정에 따라 팩토리 유형을 선택해야 합니다.

구조



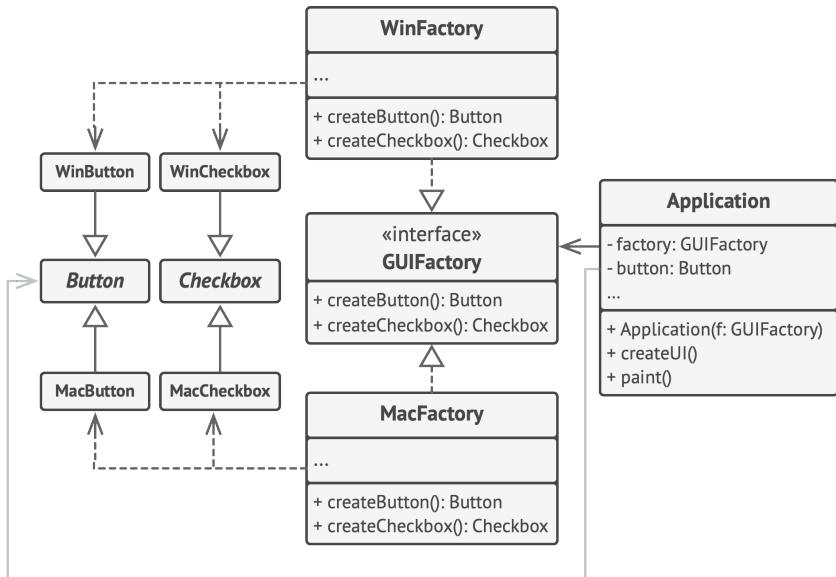
- 추상 제품들은** 제품 패밀리를 구성하는 개별 연관 제품들의 집합에 대한 인터페이스들을 선언합니다.
- 구상 제품들은** 변형들로 그룹화된 추상 제품들의 다양한 구현들입니다. 각 추상 제품(의자/소파)은 주어진 모든 변형(빅토리안/현대식)에 구현되어야 합니다.
- 추상 팩토리** 인터페이스는 각각의 추상 제품들을 생성하기 위한 여러 메서드들의 집합을 선언합니다.
- 구상 팩토리들은** 추상 팩토리의 생성 메서드들을 구현합니다. 각 구상 팩토리는 제품들의 특정 변형들에 해당하며 해당 특정 변형들만 생성합니다.

5. 구상 팩토리들은 구상 제품들을 인스턴스화하나, 그 제품들의 생성 메서드들의 시그니처들은 그에 해당하는 추상 제품들을 반환해야 합니다. 그래야 팩토리를 사용하는 클라이언트 코드가 팩토리에서 받은 제품의 특정 변형과 결합되지 않습니다. **클라이언트**는 추상 인터페이스를 통해 팩토리/제품 변형의 객체들과 소통하는 한 그 어떤 구상 팩토리/제품 변형과 작업할 수 있습니다.

의사코드

이 예시는 추상 팩토리 패턴이 크로스 플랫폼 사용자 인터페이스 (UI) 요소들을 생성하는 방법을 보여줍니다. 이 방법으로 요소들을 생성하면 클라이언트 코드는 구상 UI 클래스들과 결합하지 않으며, 모든 생성된 요소들은 선택된 운영 체제에 맞게 생성됩니다.

UI 요소들은 크로스 플랫폼 애플리케이션 내에서는 유사하게 동작할 것으로 예상되나, 다른 운영 체제 내에서는 약간씩 다르게 보일 것으로 예상됩니다. 또한 UI 요소들이 해당하는 운영 체제의 스타일과 일치하는지 확인하는 것도 당신의 책임입니다. 왜냐하면 당신은 당신의 프로그램이 윈도우에서 실행될 때 맥킨토시 컨트롤을 렌더링하는 것을 원하지 않을 것이기 때문입니다.



크로스 플랫폼 사용자 인터페이스 클래스 예시.

추상 팩토리 인터페이스는 일련의 생성 메서드들을 선언하며, 이 메서드들은 클라이언트 코드가 다양한 유형들의 UI 요소들을 생성하는 데 사용될 수 있습니다. 구상 팩토리는 특정 운영 체제에 해당하고 해당 특정 운영체제에 일치하는 UI 요소들을 생성합니다.

작동 방식은 다음과 같습니다. 앱이 시작될 때 현재 소속된 운영 체제의 유형을 확인합니다. 그 후 앱은 이 정보를 사용하여 운영 체제와 일치하는 클래스에서 팩토리 객체를 생성합니다. 나머지 코드는 이 팩토리 객체를 사용하여 UI 요소들을 만듭니다. 이렇게 하면 잘못된 요소들이 생성되는 것을 방지할 수 있습니다.

위 방법을 사용하면 클라이언트 코드가 객체의 추상 인터페이스를 통해 작업하는 한 팩토리들과 UI 요소들의 구상 클래스들에 의존하지 않게 됩니다. 또 위 방법은 클라이언트 코드가 당신이 나중에 추가할 수 있는 다른 팩토리들과 UI 요소들을 지원할 수 있도록 합니다.

이것의 결과로 새로운 변형의 UI 요소를 앱에 추가할 때마다 클라이언트 코드를 수정할 필요가 없어집니다. 이 요소들을 생성하는 새로운 팩토리 클래스를 만든 후 앱의 초기화 코드를 그 팩토리 클래스를 선택하도록 약간 수정하기만 하면 됩니다.

```

1 // 추상 팩토리 인터페이스는 다른 추상 제품들을 반환하는 메서드들의 집합을
2 // 선언합니다. 이러한 제품들을 패밀리라고 하며 이들은 상위 수준의 주제 또는
3 // 개념으로 연결됩니다. 한 가족의 제품들은 일반적으로 서로 협력할 수 있습니다.
4 // 제품들의 패밀리(제품군)에는 여러 변형이 있을 수 있지만 한 변형의 제품들은 다른
5 // 변형의 제품들과 호환되지 않습니다.
6 interface GUIFactory is
7     method Button createButton()
8     method Checkbox createCheckbox()
9
10
11 // 구상 팩토리들은 단일 변형에 속하는 제품들의 패밀리(제품군)를 생성합니다. 이
12 // 팩토리는 결과 제품들의 호환을 보장합니다. 구상 팩토리 메서드의 시그니처들은 추상
13 // 제품을 반환하는 반면, 메서드 내부에서는 구상 제품이 인스턴스화됩니다.
14 class WinFactory implements GUIFactory is
15     method Button createButton() is
16         return new WinButton()
17     method Checkbox createCheckbox() is
18         return new WinCheckbox()
```

```
19
20 // 각 구상 팩토리에는 해당하는 제품 변형이 있습니다.
21 class MacFactory implements GUIFactory {
22     method createButton():Button is
23         return new MacButton()
24     method createCheckbox():Checkbox is
25         return new MacCheckbox()
26
27
28 // 제품 패밀리의 각 개별 제품에는 기초 인터페이스가 있어야 합니다. 이 제품의 모든
29 // 변형은 이 인터페이스를 구현해야 합니다.
30 interface Button is
31     method paint()
32
33 // 구상 제품들은 해당하는 구상 팩토리에서 생성됩니다.
34 class WinButton implements Button {
35     method paint() is
36         // 버튼을 윈도우 스타일로 렌더링하세요.
37
38 class MacButton implements Button {
39     method paint() is
40         // 버튼을 맥 스타일로 렌더링하세요.
41
42 // 다음은 다른 제품의 기초 인터페이스입니다. 모든 제품은 상호 작용할 수 있지만 같은
43 // 구상 변형의 제품들 사이에서만 적절한 상호 작용이 가능합니다.
44 interface Checkbox is
45     method paint()
46
47 class WinCheckbox implements Checkbox {
48     method paint() is
49         // 윈도우 스타일의 확인란을 렌더링하세요.
50
```

```

51  class MacCheckbox implements Checkbox is
52      method paint() is
53          // 맥 스타일의 확인란을 렌더링하세요.
54
55
56  // 클라이언트 코드는 GUIFactory, Button 및 Checkbox와 같은 추상 유형을
57  // 통해서만 팩토리를 및 제품들과 작동하며, 이는 클라이언트 코드를 손상하지 않고
58  // 클라이언트 코드에 모든 팩토리 또는 하위 클래스를 전달할 수 있게 해줍니다.
59  class Application is
60      private field factory: GUIFactory
61      private field button: Button
62      constructor Application(factory: GUIFactory) is
63          this.factory = factory
64      method createUI() is
65          this.button = factory.createButton()
66      method paint() is
67          button.paint()
68
69
70  // 앱은 현재 설정 또는 환경 설정에 따라 팩토리 유형을 선택한 후 팩토리를 런타임
71  // 때(일반적으로는 초기화 단계에서) 생성합니다.
72  class ApplicationConfigurator is
73      method main() is
74          config = readApplicationConfigFile()
75
76          if (config.OS == "Windows") then
77              factory = new WinFactory()
78          else if (config.OS == "Mac") then
79              factory = new MacFactory()
80          else
81              throw new Exception("Error! Unknown operating system.")
82

```

83 Application app = new Application(factory)

💡 적용

- ❖ 추상 팩토리는 당신의 코드가 관련된 제품군의 다양한 패밀리들과 작동해야 하지만 해당 제품들의 구상 클래스들에 의존하고 싶지 않을 때 사용하세요. 왜냐하면 이러한 클래스들은 당신에게 미리 알려지지 않았을 수 있으며, 그 때문에 향후 확장성(extensibility)을 허용하기를 원할 수 있기 때문입니다.
- ❖ 추상 팩토리는 제품 패밀리의 각 클래스에서부터 객체들을 생성할 수 있는 인터페이스를 제공합니다. 위 인터페이스를 통해 코드가 객체들을 생성하는 한 당신은 당신의 앱에서 이미 생성된 제품들과 일치하지 않는 잘못된 제품 변형을 생성하지 않을지 걱정할 필요가 없습니다.
- ❖ 코드에 클래스가 있고, 이 클래스의 팩토리 메서드들의 집합의 기본 책임이 뚜렷하지 않을 때 추상 팩토리 구현을 고려하세요.
- ❖ 잘 설계된 프로그램에서는 각 클래스는 하나의 책임만 가집니다. 클래스가 여러 제품 유형을 상대할 경우, 클래스의 팩토리 메서드들을 독립실행형 팩토리 클래스 또는 완전한 추상 팩토리 구현으로 추출할 가치가 있을 수 있습니다.

▣ 구현방법

1. 고유한 제품 유형들 대 변형 제품들을 나타내는 매트릭스를 매핑하세요.
2. 모든 제품 변형들에 대한 추상 제품 인터페이스들을 선언하세요. 그 후 모든 구상 제품 클래스들이 위 인터페이스들을 구현하도록 하세요.
3. 추상 팩토리 인터페이스를 모든 추상 제품들에 대한 생성 메서드들의 집합과 함께 선언하세요.
4. 각 제품 변형에 대해 각각 하나의 구상 팩토리 클래스 집합을 구현하세요.
5. 앱 어딘가에 팩토리 초기화 코드를 생성하세요. 초기화 코드는 앱 설정 또는 현재 환경에 따라 구상 팩토리 클래스 중 하나를 인스턴스화해야 합니다. 이 팩토리 객체를 제품을 생성하는 모든 클래스들에 전달하세요.
6. 코드를 검토해서 제품 생성자에 대한 모든 직접 호출을 찾으세요. 이 호출들을 팩토리 객체에 대한 적절한 생성 메서드에 대한 호출들로 교체하세요.

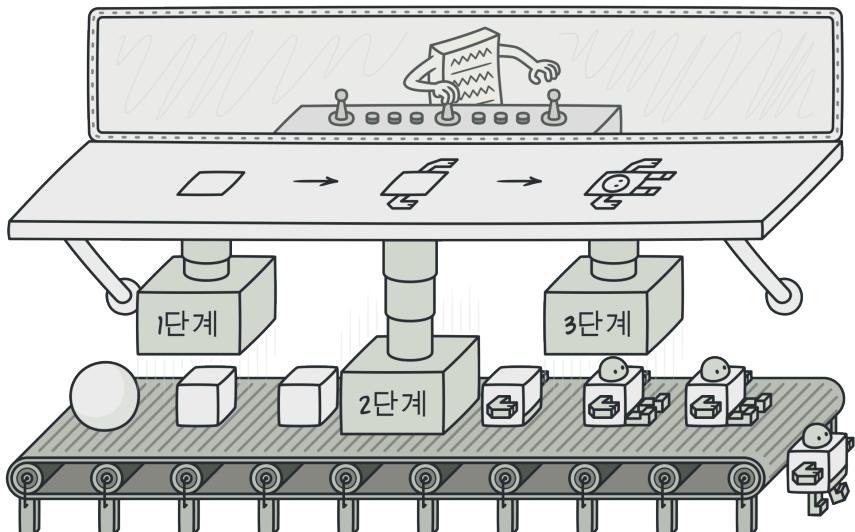
⚠ 장단점

- ✓ 팩토리에서 생성되는 제품들의 상호 호환을 보장할 수 있습니다.
- ✓ 구상 제품들과 클라이언트 코드 사이의 단단한 결합을 피할 수 있습니다.
- ✓ 단일 책임 원칙. 제품 생성 코드를 한 곳으로 추출하여 코드를 더 쉽게 유지보수할 수 있습니다.
- ✓ 개방/폐쇄 원칙. 기존 클라이언트 코드를 훼손하지 않고 제품의 새로운 변형들을 생성할 수 있습니다.
- ✗ 패턴과 함께 새로운 인터페이스들과 클래스들이 많이 도입되기 때문에 코드가 필요 이상으로 복잡해질 수 있습니다.

↔ 다른 패턴과의 관계

- 많은 디자인은 복잡성이 낮고 자식 클래스들을 통해 더 많은 커스터마이징이 가능한 팩토리 메서드로 시작해 더 유연하면서도 더 복잡한 추상 팩토리, 프로토타입 또는 빌더 패턴으로 발전해 나갑니다.
- 빌더는 복잡한 객체들을 단계별로 생성하는 데 중점을 둡니다. 추상 팩토리는 관련된 객체들의 패밀리들을 생성하는 데 중점을 둡니다. 추상 팩토리는 제품을 즉시 반환하지만 빌더는 제품을 가져오기 전에 당신이 몇 가지 추가 생성 단계들을 실행할 수 있도록 합니다.

- **추상 팩토리** 클래스들은 **팩토리 메서드**들의 집합을 기반으로 하는 경우가 많습니다. 그러나 당신은 또한 **프로토타입**을 사용하여 **추상 팩토리**의 구상 클래스들의 생성 메서드들을 구현할 수도 있습니다.
- **추상 팩토리**는 하위시스템 객체들이 클라이언트 코드에서 생성되는 방식만 숨기고 싶을 때 **퍼사드** 대신 사용할 수 있습니다.
- 당신은 **추상 팩토리를 브리지**와 함께 사용할 수 있습니다. 이 조합은 브리지에 의해 정의된 어떤 추상화들이 특정 구현들과만 작동할 수 있을 때 유용합니다. 이런 경우에 **추상 팩토리**는 이러한 관계들을 캡슐화하고 클라이언트 코드에서부터 복잡성을 숨길 수 있습니다.
- **추상 팩토리들, 빌더들 및 프로토타입들은 모두 싱글턴으로 구현할 수 있습니다.**



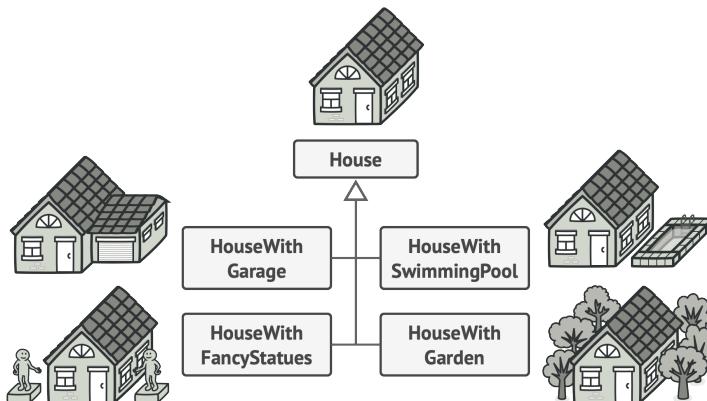
빌더 패턴

다음 이름으로도 불립니다: Builder

빌더는 복잡한 객체들을 단계별로 생성할 수 있도록 하는 생성 디자인 패턴입니다. 이 패턴을 사용하면 같은 제작 코드를 사용하여 객체의 다양한 유형들과 표현을 제작할 수 있습니다.

(:(문제

많은 필드와 중첩된 객체들을 힘들게 단계별로 초기화해야 하는 복잡한 객체를 상상해 보세요. 이러한 초기화 코드는 일반적으로 많은 매개변수가 있는 괴물 같은 생성자 내부에 묻혀 있습니다. 또, 더 최악의 상황에는 클라이언트 코드 전체에 흩어져 있을 수도 있습니다.

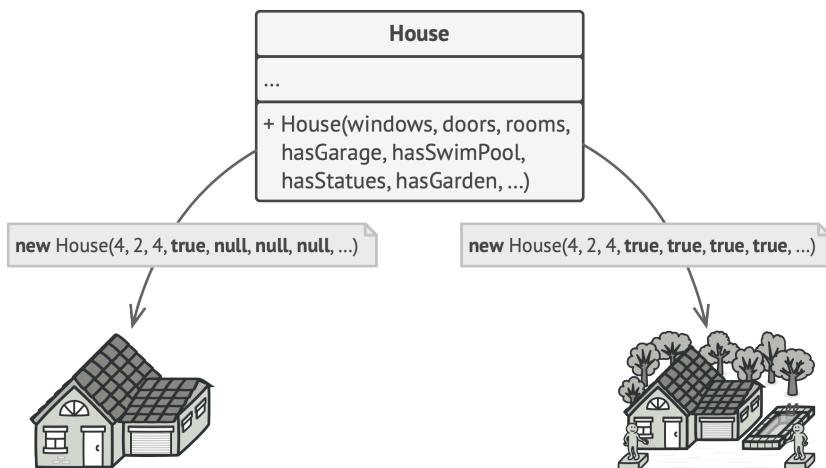


당신은 객체의 가능한 모든 설정에 자식 클래스를 만들어 프로그램을 매우 복잡하게 만들 수 있습니다.

예를 들어 `House` (집) 객체를 만드는 방법에 대해 생각해 봅시다. 간단한 집을 지으려면 네 개의 벽과 바닥을 만든 후 문도 설치하고 한 쌍의 창문도 맞춘 후 지붕도 만들어야 합니다. 하지만 뒤틀과 기타 물품(난방 시스템, 배관 및 전기 배선 등)이 있는 더 크고 현대적인 집을 원하면 어떻게 해야 할까요?

위 문제의 가장 간단한 해결책은 기초 `House` 클래스를 확장하고 매개변수의 모든 조합을 포함하는 자식 클래스들의 집합을 만드는 것입니다. 그러나 당신은 결국 상당한 수의 자식 클래스를 만들게 될 것입니다. 새로운 매개변수(예: 현관 스타일)를 추가할 때마다 이 계층구조는 훨씬 더 복잡해질 것입니다.

자식 클래스들을 늘리지 않는 다른 접근 방식이 있습니다. 기초 `House` 클래스에 `House` 객체를 제어하는 모든 가능한 매개변수를 포함한 거대한 생성자를 만드는 것입니다. 이 접근 방식은 실제로 자식 클래스들의 필요성을 제거하나, 다른 문제를 만들어 냅니다.



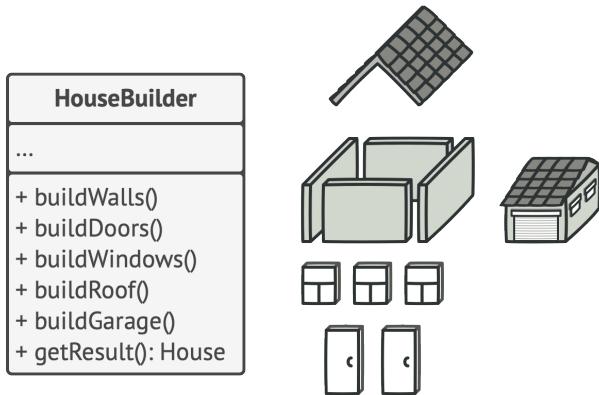
매개변수가 많은 생성자의 단점은 모든 매개변수가 항상 필요한 것은 아니라는 점입니다.

보통 대부분의 매개변수가 사용되지 않아 생성자 호출들의 코드가 매우 못생겨질 것입니다. 예를 들어, 극소수의 집들에만 수영장이

있으므로 수영장과 관련된 매개변수들은 십중팔구 사용되지 않을 것입니다.

😊 해결책

빌더 패턴은 자신의 클래스에서 객체 생성 코드를 추출하여 *builders*(건축업자들)라는 별도의 객체들로 이동하도록 제안합니다.



빌더 패턴은 복잡한 객체들을 단계별로 생성할 수 있도록 합니다. 빌더는 제품이 생성되는 동안 다른 객체들이 제품에 접근(access)하는 것을 허용하지 않습니다.

이 패턴은 객체 생성을 일련의 단계들(`buildWalls` (벽 건설), `buildDoor` (문 건설) 등)로 정리하며, 객체를 생성하고 싶으면 위 단계들을 *builder*(빌더) 객체에 실행하면 됩니다. 또 중요한 점은 모든 단계를 호출할 필요가 없다는 것으로, 객체의 특정 설정을 제작하는 데 필요한 단계들만 호출하면 됩니다.

일부 건축 단계들은 제품의 다양한 표현을 건축해야 하는 경우 다른 구현들이 필요할 수 있습니다. 예를 들어, 오두막의 벽은 나무로 지을 수 있지만 성벽은 돌로 지어야 합니다.

이런 경우 같은 건축 단계들의 집합을 다른 방식으로 구현하는 여러 다른 빌더 클래스를 생성할 수 있으며, 그런 다음 건축 프로세스(즉, 건축 단계에 대한 순서화된 호출들의 집합)내에서 이러한 빌더들을 사용하여 다양한 종류의 객체를 생성할 수 있습니다.



다양한 빌더들은 다양한 방식으로 같은 작업을 실행합니다.

예를 들어, 나무와 유리로 모든 것을 건축하는 건축가, 돌과 철로 모든 것을 건축하는 두 번째 건축가, 금과 다이아몬드로 모든 것을 건축하는 세 번째 건축가가 있다고 상상해 보세요. 이 세 건축가에 대해 같은 단계들의 집합을 호출하면 첫 번째 건축업자에게서부터는 일반 주택을, 두 번째 건축업자에게서부터는 작은 성을, 세 번째 건축업자에게서부터는

궁전을 얻습니다. 그러나 위에 예시된 경우는 건축 단계들을 호출하는 클라이언트 코드가 공통 인터페이스를 사용하여 빌더들과 상호 작용할 수 있는 경우에만 작동합니다.

디렉터 (관리자)

더 나아가 제품을 생성하는 데 사용하는 빌더 단계들에 대한 일련의 호출을 디렉터 (관리자)라는 별도의 클래스로 추출할 수 있습니다. 디렉터 클래스는 제작 단계들을 실행하는 순서를 정의하는 반면 빌더는 이러한 단계들에 대한 구현을 제공합니다.



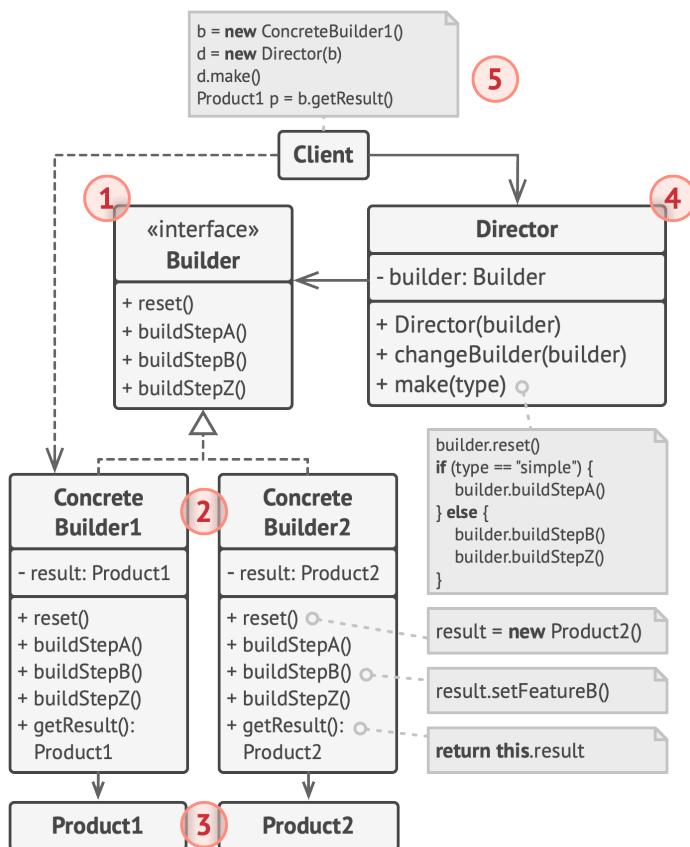
디렉터는 작동하는 제품을 얻기 위하여 어떤 건축 단계들을 실행해야 하는지 알고 있습니다.

프로그램에 디렉터 클래스를 포함하는 것은 필수사항은 아닙니다. 당신은 언제든지 클라이언트 코드에서 생성 단계들을 직접 특정 순서로 호출할 수 있습니다. 그러나 디렉터 클래스는 다양한 생성

루틴들을 배치하여 프로그램 전체에서 재사용할 수 있는 좋은 장소가 될 수 있습니다.

또한 디렉터 클래스는 클라이언트 코드에서 제품 생성의 세부 정보를 완전히 숨깁니다. 클라이언트는 빌더를 디렉터와 연관시키고 디렉터와 생성을 시행한 후 빌더로부터 결과를 얻기만 하면 됩니다.

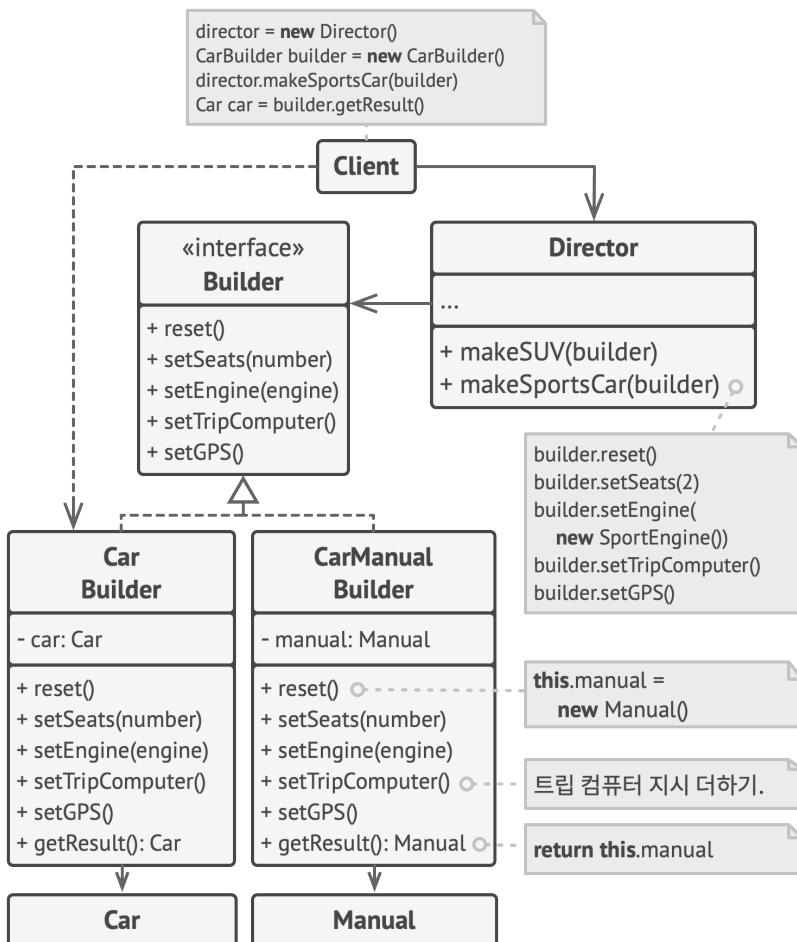
구조



1. **빌더** 인터페이스는 모든 유형의 빌더들에 공통적인 제품 생성 단계들을 선언합니다.
2. **구상 빌더**들은 생성 단계들의 다양한 구현을 제공합니다. 또 구상 빌더들은 공통 인터페이스를 따르지 않는 제품들도 생산할 수 있습니다.
3. **제품들은** 그 결과로 나온 객체들입니다. 다른 빌더들에 의해 생성된 제품들은 같은 클래스 계층구조 또는 인터페이스에 속할 필요가 없습니다.
4. **디렉터** 클래스는 생성 단계들을 호출하는 순서를 정의하므로 제품들의 특정 설정을 만들고 재사용할 수 있습니다.
5. **클라이언트**는 빌더 객체들 중 하나를 디렉터와 연결해야 합니다. 일반적으로 위 연결은 디렉터 생성자의 매개변수들을 통해 한번만 수행되며, 그 후 디렉터는 모든 추가 생성에 해당 빌더 객체들을 사용합니다. 그러나 클라이언트가 빌더 객체를 디렉터의 프로덕션 메서드에 전달할 때를 위한 대안적 접근 방식이 있습니다. 이 경우 디렉터와 함께 무언가를 만들 때마다 다른 빌더를 사용할 수 있습니다.

의사코드

아래 **빌더** 패턴 예시는 자동차와 같은 다양한 유형의 제품들을 생성할 때 동일한 객체 생성 코드를 재사용하고 그에 해당하는 매뉴얼을 만드는 방법을 보여줍니다.



자동차들의 단계별 생성과 해당 자동차 모델들에 맞는 사용자 설명서들의 예시.

자동차는 수백 가지 방법으로 생성될 수 있는 복잡한 객체입니다. 거대한 생성자로 `Car` (자동차) 클래스를 부풀리는 대신, 우리는 자동차 조립 코드를 별도의 자동차 빌더 클래스로 추출했습니다. 이 클래스에는 자동차의 다양한 부분들을 설정하기 위한 메서드들의 집합이 있습니다.

클라이언트 코드가 미세하게 조정된 특별한 자동차 모델을 조립해야 하는 경우 빌더와 직접 작동할 수 있습니다. 반면에 클라이언트는 조립을 디렉터 클래스로 위임할 수 있으며, 이 디렉터 클래스는 빌더를 사용하여 가장 인기 있는 자동차 모델들을 조립하는 방법들을 알고 있습니다.

모든 자동차에는 사용설명서가 필요합니다. 사용설명서에는 해당 자동차의 모든 기능이 설명되어 있으므로 설명서의 세부 사항은 모델마다 다릅니다. 그러므로 실제 자동차들과 그들의 해당 사용설명서들 모두에 기존 제작 프로세스들을 재사용하는 것이 합리적입니다. 물론 사용설명서를 만드는 것은 자동차를 만드는 것과 같지 않기 때문에 사용설명서 작성을 전문으로 하는 또 다른 빌더 클래스를 제공해야 합니다. 이 클래스는 자동차 제작 관련 형제 클래스와 같은 제작 메서드들을 구현하지만, 자동차 부품들을 제작하는 대신 설명합니다. 이러한 빌더들을 같은 디렉터 객체에 전달하여 자동차 또는 설명서를 제작할 수 있습니다.

마지막 부분은 결과 객체를 가져오는 것입니다. 금속으로 만들어진 자동차와 종이 사용설명서는 관련은 있지만 여전히 매우 다른 사물들입니다. 디렉터를 구상 제품 클래스들에 결합하지 않고 디렉터에 결과를 가져오는 메서드를 배치할 수는 없습니다. 따라서 우리는 작업을 수행한 빌더로부터 생성의 결과를 얻습니다.

```

1 // 빌더 패턴을 사용하는 것은 제품에 매우 복잡하고 광범위한 설정이 필요한 경우에만
2 // 의미가 있습니다. 다음 두 제품은 공통 인터페이스는 없지만 관련되어 있습니다.
3 class Car is
4     // 자동차에는 GPS, 트립 컴퓨터 및 몇 개의 좌석이 있을 수 있습니다. 다른
5     // 모델의 자동차(스포츠카, SUV, 오픈카)에는 다른 기능들이 설치되거나
6     // 활성화되어 있을 수 있습니다.
7
8 class Manual is
9     // 각 자동차에는 자동차의 설정에 해당하는, 모든 기능을 설명하는 사용 설명서가
10    // 있어야 합니다.
11
12
13 // 빌더 인터페이스는 제품 객체들의 다른 부분들을 만드는 메서드들을 지정합니다.
14 interface Builder is
15     method reset()
16     method setSeats(...)
17     method setEngine(...)
18     method setTripComputer(...)
19     method setGPS(...)

20
21 // 구상 빌더 클래스들은 빌더 인터페이스를 따르고 빌드 단계들의 특정 구현들을
22 // 제공합니다. 당신의 프로그램에는 각기 다르게 구현된 여러 가지 빌더 변형들이 있을
23 // 수 있습니다.

```

```
24 class CarBuilder implements Builder is
25     private field car:Car
26
27     // 새로운 빌더 인스턴스에는 인스턴스가 추가적인 조립과정에서 사용하는 빈 제품
28     // 객체가 포함되어야 합니다.
29     constructor CarBuilder() is
30         this.reset()
31
32     // reset 메서드는 구축 중인 객체를 지웁니다.
33     method reset() is
34         this.car = new Car()
35
36     // 모든 생성 단계들은 같은 제품의 인스턴스와 작동합니다.
37     method setSeats(...) is
38         // 차량의 좌석 수를 설정하세요.
39
40     method setEngine(...) is
41         // 해당 엔진을 설치하세요.
42
43     method setTripComputer(...) is
44         // 트립 컴퓨터를 설치하세요.
45
46     method setGPS(...) is
47         // GPS를 설치하세요.
48
49     // 구상 빌더들은 결과들을 가져오기 위한 자체 메서드들을 제공해야 합니다.
50     // 왜냐하면 다양한 유형의 빌더들은 모두 같은 인터페이스를 따르지 않는 완전히
51     // 다른 제품들을 생성할 수 있기 때문입니다. 따라서 이러한 메서드는 빌더
52     // 인터페이스에서 선언할 수 없습니다. 적어도 이는 정적 타입 언어에서는
53     // 불가능합니다.
54     //
55     // 최종 결과를 클라이언트에 반환한 후 일반적으로 빌더 인스턴스는 다른 제품
```

```
56 // 생산을 시작할 준비가 되어 있을 것이라고 예상됩니다. 이것이  
57 // `getProduct` 메서드의 본문 끝에서 reset 메서드를 호출하는 것이  
58 // 일반적인 관행인 이유입니다. 하지만 반드시 이렇게 해야 하는 것은  
59 // 아니라서, 빌더가 클라이언트 코드로부터 명시적으로 reset 호출을 받을  
60 // 때까지 이전 결과를 삭제하지 않고 기다리게 만들 수 있습니다.  
61 method getProduct():Car is  
62     product = this.car  
63     this.reset()  
64     return product  
65  
66 // 다른 생성 패턴과 달리 빌더를 사용하면 공통 인터페이스를 따르지 않는 제품들을  
67 // 생성할 수 있습니다.  
68 class CarManualBuilder implements Builder is  
69     private field manual:Manual  
70  
71 constructor CarManualBuilder() is  
72     this.reset()  
73  
74 method reset() is  
75     this.manual = new Manual()  
76  
77 method setSeats(...) is  
78     // 자동차 좌석의 기능들을 문서화하세요.  
79  
80 method setEngine(...) is  
81     // 엔진 사용 지침을 추가하세요.  
82  
83 method setTripComputer(...) is  
84     // 트립 컴퓨터 사용 지침을 추가하세요.  
85  
86 method setGPS(...) is  
87     // GPS 사용 지침을 추가하세요.
```

```
88
89     method getProduct():Manual is
90         // 매뉴얼을 반환하고 빌더를 초기화하세요.
91
92
93     // �렉터는 특정 순서로 생성 단계들을 실행하는 책임만 있습니다. 이것은 특정 순서나
94     // 설정에 따라 제품들을 생성할 때 유용합니다. 엄밀히 말하면, 클라이언트가 빌더들을
95     // 직접 제어할 수 있으므로 �렉터 클래스는 선택 사항입니다.
96
97     class Director is
98         // �렉터는 클라이언트 코드가 전달하는 모든 빌더 인스턴스와 함께 작동합니다.
99         // 그러면 클라이언트 코드는 새로 조립된 제품의 최종 유형을 변경할 수
100        // 있습니다. �렉터는 같은 생성 단계들을 사용하여 여러 제품 변형들을 생성할
101        // 수 있습니다.
102
103     method constructSportsCar(builder: Builder) is
104         builder.reset()
105         builder.setSeats(2)
106         builder.setEngine(new SportEngine())
107         builder.setTripComputer(true)
108         builder.setGPS(true)
109
110
111
112     // 클라이언트 코드는 빌더 객체를 만든 후 이를 딕션너리에게 전달한 다음 생성
113     // 프로세스를 시작합니다. 최종 결과는 빌더 객체에서 가져옵니다.
114
115     class Application is
116         method makeCar() is
117             director = new Director()
118
119             CarBuilder builder = new CarBuilder()
```

```
120     director.constructSportsCar(builder)
121     Car car = builder.getProduct()
122
123     CarManualBuilder builder = new CarManualBuilder()
124     director.constructSportsCar(builder)
125
126     // 디렉터는 구상 빌더들 및 제품들에 의존하지 않고 인식하지 못하기 때문에
127     // 최종 제품은 종종 빌더 객체에서 가져옵니다.
128     Manual manual = builder.getProduct()
```

💡 적용

💡 '점총적 생성자'를 제거하기 위하여 빌더 패턴을 사용하세요.

👉 10개의 선택적 매개변수가 있는 생성자가 있다고 가정합니다. 이렇게 복잡한 생성자를 호출하는 것은 매우 불편합니다. 따라서 이 생성자를 오버로드하고 더 적은 수의 매개변수들을 사용하는 더 짧은 생성자 버전들을 여러 개 만듭니다. 이러한 생성자들은 여전히 주 생성자를 참조하며, 생략된 매개변수들에 일부 기본값들을 전달합니다.

```

1 class Pizza {
2     Pizza(int size) { ... }
3     Pizza(int size, boolean cheese) { ... }
4     Pizza(int size, boolean cheese, boolean pepperoni) { ... }
5     // ...

```

이렇게 복잡한 생성자를 만드는 것은 C#나 자바와 같이 메서드 오버로딩을 지원하는 언어들에서만 가능합니다.

빌더 패턴을 사용하면 실제로 필요한 단계들만 사용하여 단계별로 객체들을 생성할 수 있으며, 패턴을 구현한 후에는 더 이상 수십 개의 매개변수를 생성자에 집어넣을 필요가 없습니다.

 **빌더 패턴은 당신의 코드가 일부 제품의 다른 표현들(예: 석조 및 목조 주택들)을 생성할 수 있도록 하고 싶을 때 사용하세요.**

 **빌더 패턴은 제품의 다양한 표현의 생성 과정이 세부 사항만 다른 유사한 단계를 포함할 때 적용할 수 있습니다.**

기초 빌더 인터페이스는 가능한 모든 생성 단계들을 정의하고 구상 빌더들은 이러한 단계들을 구현하여 제품의 여러 표현을 생성합니다. 또 한편 디렉터 클래스는 건설 순서를 안내합니다.

 **빌더를 사용하여 복합체 트리들 또는 기타 복잡한 객체들을 생성하세요.**

- ↳ 빌더 패턴을 사용하면 제품들을 단계별로 생성할 수 있으며, 또 최종 제품을 손상하지 않고 일부 단계들의 실행을 연기할 수 있습니다. 그리고 재귀적으로 단계들을 호출할 수도 있는데, 이는 객체 트리를 구축해야 할 때 매우 유용합니다.

빌더는 생성 단계들을 수행하는 동안 미완성 제품을 노출하지 않으며, 이는 클라이언트 코드가 불완전한 결과를 가져오는 것을 방지합니다.

▣ 구현방법

1. 사용할 수 있는 모든 제품 표현을 생성하기 위한 공통 생성 단계들을 명확하게 정의할 수 있는지 확인하세요. 그렇게 하지 못하면, 패턴 구현을 진행할 수 없습니다.
2. 기초 빌더 인터페이스에서 이 단계를 선언하세요.
3. 각 제품 표현에 대한 구상 빌더 클래스를 만들고 해당 생성 단계들을 구현하세요.

생성 결과를 가져오는 메서드를 구현하는 것을 잊지 마세요. 빌더 인터페이스 내에서 이 메서드를 선언할 수 없는 이유는 다양한 빌더들이 공통 인터페이스가 없는 제품들을 생성할 수 있기 때문입니다. 따라서 이러한 메서드의 반환 유형이 무엇인지 알 수 없습니다. 그러나 단일 계층구조의 제품들을 처리하는 경우 생성

결과를 가져오는 메서드를 기초 인터페이스에 안전하게 추가할 수 있습니다.

4. 디렉터 클래스를 만드는 것에 대해 생각해보세요. 같은 빌더 객체를 사용하여 제품을 제작하는 다양한 방법을 캡슐화할 수 있습니다.
5. 클라이언트 코드는 빌더 객체들과 디렉터 객체들을 모두 생성합니다. 제작이 시작되기 전에 클라이언트는 빌더 객체를 디렉터에게 전달해야 합니다. 일반적으로 클라이언트는 디렉터의 클래스 생성자의 매개변수들을 통해 위 작업을 한 번만 수행합니다. 그 후 디렉터는 모든 추가 제작에서 빌더 객체를 사용합니다. 또 대안적인 접근 방식이 있는데, 이 방식은 빌더가 디렉터의 특정 제품 제작 메서드에 전달되는 것입니다.
6. 모든 제품이 같은 인터페이스를 따르는 경우에만 디렉터로부터 직접 생성 결과를 얻을 수 있습니다. 그렇지 않으면 클라이언트는 빌더에서 결과를 가져와야 합니다.

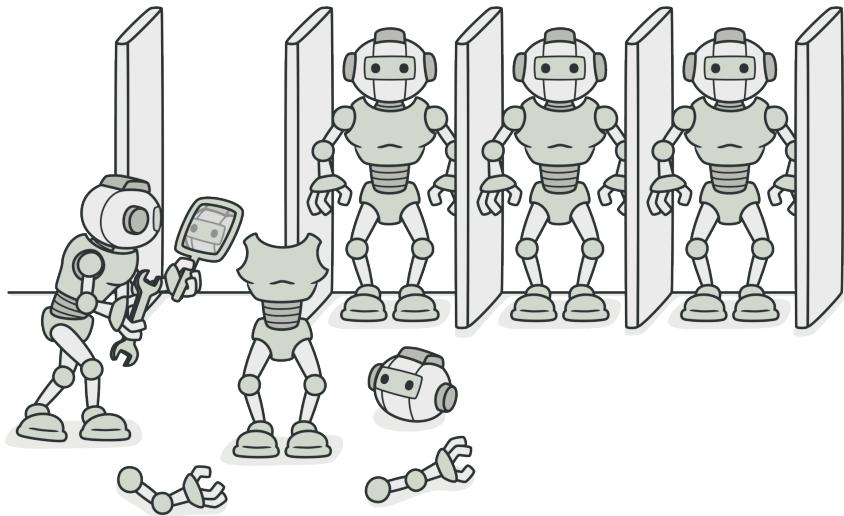
⚠️ 장단점

- ✓ 객체들을 단계별로 생성하거나 생성 단계들을 연기하거나 재귀적으로 단계들을 실행할 수 있습니다.
- ✓ 제품들의 다양한 표현을 만들 때 같은 생성 코드를 재사용할 수 있습니다.

- ✓ 단일 책임 원칙. 제품의 비즈니스 로직에서 복잡한 생성 코드를 고립시킬 수 있습니다.
- ✗ 패턴이 여러 개의 새 클래스들을 생성해야 하므로 코드의 전반적인 복잡성이 증가합니다.

↔ 다른 패턴과의 관계

- 많은 디자인은 복잡성이 낮고 자식 클래스들을 통해 더 많은 커스터마이징이 가능한 팩토리 메서드로 시작해 더 유연하면서도 더 복잡한 추상 팩토리, 프로토타입 또는 빌더 패턴으로 발전해 나갑니다.
- 빌더는 복잡한 객체들을 단계별로 생성하는 데 중점을 둡니다. 추상 팩토리는 관련된 객체들의 패밀리들을 생성하는 데 중점을 둡니다. 추상 팩토리는 제품을 즉시 반환하지만 빌더는 제품을 가져오기 전에 당신이 몇 가지 추가 생성 단계들을 실행할 수 있도록 합니다.
- 당신은 복잡한 복합체 패턴 트리를 생성할 때 빌더를 사용할 수 있습니다. 왜냐하면 빌더의 생성 단계들을 재귀적으로 작동하도록 프로그래밍할 수 있기 때문입니다.
- 빌더를 브리지와 조합할 수 있습니다. 딕션 퍼스는 추상화의 역할을 하고 다양한 빌더들은 구현의 역할을 합니다.
- 추상 팩토리들, 빌더들 및 프로토타입들은 모두 싱글턴으로 구현할 수 있습니다.



프로토타입 패턴

다음 이름으로도 불립니다: 클론, Prototype

프로토타입은 코드를 그들의 클래스들에 의존시키지 않고 기존 객체들을 복사할 수 있도록 하는 생성 디자인 패턴입니다.

(:() 문제

객체가 있고 그 객체의 정확한 복사본을 만들고 싶다고 가정하면, 어떻게 하시겠습니까? 먼저 같은 클래스의 새 객체를 생성해야 합니다. 그런 다음 원본 객체의 모든 필드들을 살펴본 후 해당 값을 새 객체에 복사해야 합니다.

너무 쉽군요! 하지만 함정이 있습니다. 객체의 필드들 중 일부가 비공개여서 객체 자체의 외부에서 볼 수 없을 수 있으므로 모든 객체를 그런 식으로 복사하지 못합니다.



객체를 '외부에서부터' 복사하는 것은 항상 가능하지 않습니다.

이 직접적인 접근 방식에는 한 가지 문제가 더 있습니다. 객체의 복제본을 생성하려면 객체의 클래스를 알아야 하므로, 당신의 코드가 해당 클래스에 의존하게 된다는 것입니다. 또, 예를 들어 메서드의 매개변수가 일부 인터페이스를 따르는 모든 객체를

수락할 때 당신은 그 객체가 따르는 인터페이스만 알고, 그 객체의 구상 클래스는 알지 못할 수 있습니다.

😊 해결책

프로토타입 패턴은 실제로 복제되는 객체들에 복제 프로세스를 위임합니다. 패턴은 복제를 지원하는 모든 객체에 대한 공통 인터페이스를 선언합니다. 이 인터페이스를 사용하면 코드를 객체의 클래스에 결합하지 않고도 해당 객체를 복제할 수 있습니다. 일반적으로 이러한 인터페이스에는 단일 `clone` 메서드만 포함됩니다.

`clone` 메서드의 구현은 모든 클래스에서 매우 유사합니다. 이 메서드는 현재 클래스의 객체를 만든 후 이전 객체의 모든 필드 값을 새 객체로 전달합니다. 대부분의 프로그래밍 언어는 객체들이 같은 클래스에 속한 다른 객체의 비공개 필드들에 접근 (access) 할 수 있도록 하므로 비공개 필드들을 복사하는 것도 가능합니다.

복제를 지원하는 객체를 [•][•][•][•][•] [•]프로토타입이라고 합니다. 당신의 객체들에 수십 개의 필드와 수백 개의 가능한 설정들이 있는 경우 이를 복제하는 것이 서브클래싱의 대안이 될 수 있습니다.

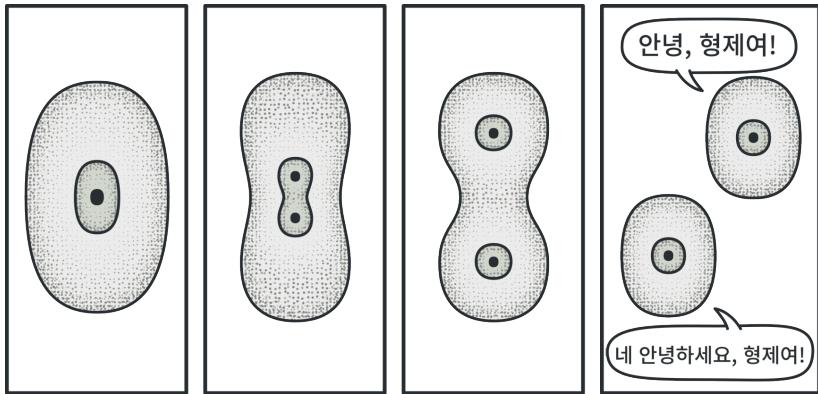


미리 만들어진 프로토타입은 서브클래싱의 대안이 될 수 있습니다.

프로토타이핑은 다음과 같이 작동합니다. 일단, 다양한 방식으로 설정된 객체들의 집합을 만듭니다. 그 후 설정한 것과 비슷한 객체가 필요할 경우 처음부터 새 객체를 생성하는 대신 프로토타입을 복제하면 됩니다.

☞ 실제상황 적용

실제 산업에서의 프로토타입(원기)은 제품의 대량 생산을 시작하기 전에 다양한 테스트를 수행하는 데 사용됩니다. 그러나 프로그래밍의 프로토타입의 경우 프로토타입들은 실제 생산과정에 참여하지 않고 대신 수동적인 역할을 합니다.

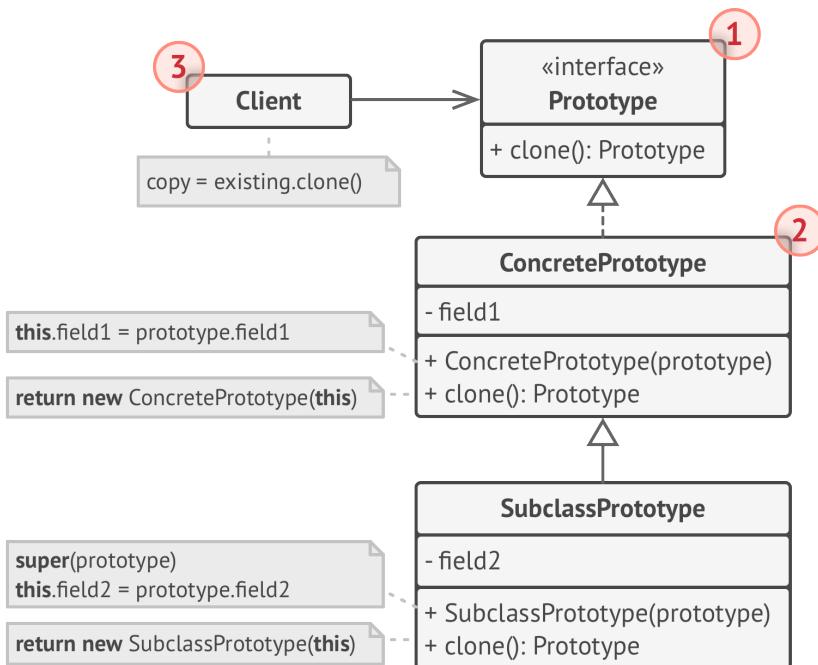


세포의 분열

산업 프로토타입들은 실제로 자신을 복제하지 않기 때문에, 프로토타입 패턴에 더 가까운 예시는 세포의 유사분열 과정입니다. 유사분열 후에는 한 쌍의 같은 세포가 형성됩니다. 원본 세포는 프로토타입 역할을 하며 복사본을 만드는 데 능동적인 역할을 합니다.

구조

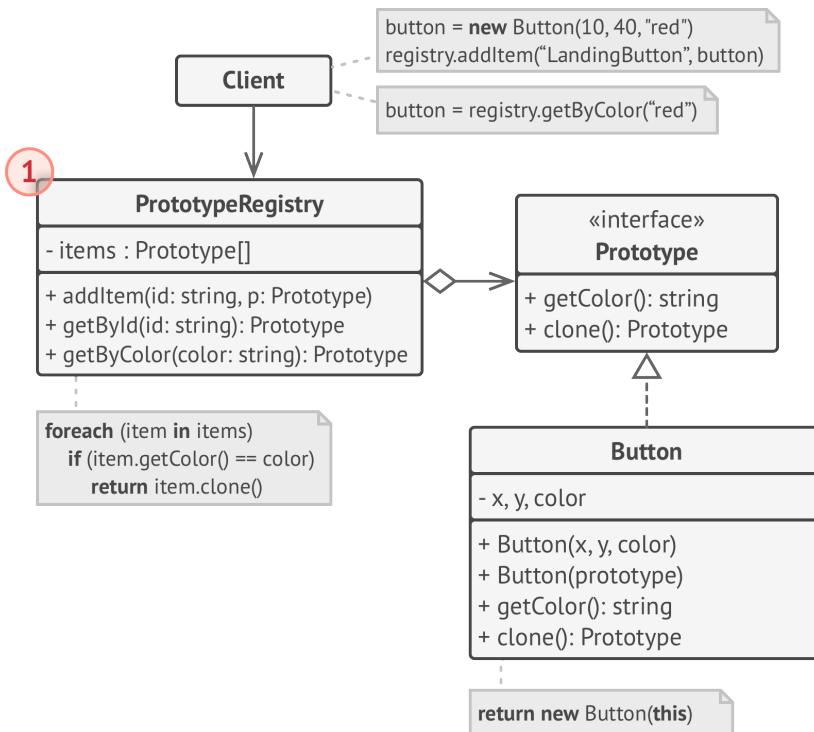
기초 구현



1. **프로토타입** 인터페이스는 복제 메서드들을 선언하며, 이 메서드들의 대부분은 단일 `clone` 메서드입니다.
2. **구상 프로토타입** 클래스는 복제 메서드를 구현합니다. 원본 객체의 데이터를 복제본에 복사하는 것 외에도 이 메서드는 복제 프로세스와 관련된 일부 예외적인 경우들도 처리할 수도 있습니다. (예: 연결된 객체 복제, 재귀 종속성 풀기).

3. **클라이언트**는 프로토타입 인터페이스를 따르는 모든 객체의 복사본을 생성할 수 있습니다.

프로토타입 레지스트리 구현

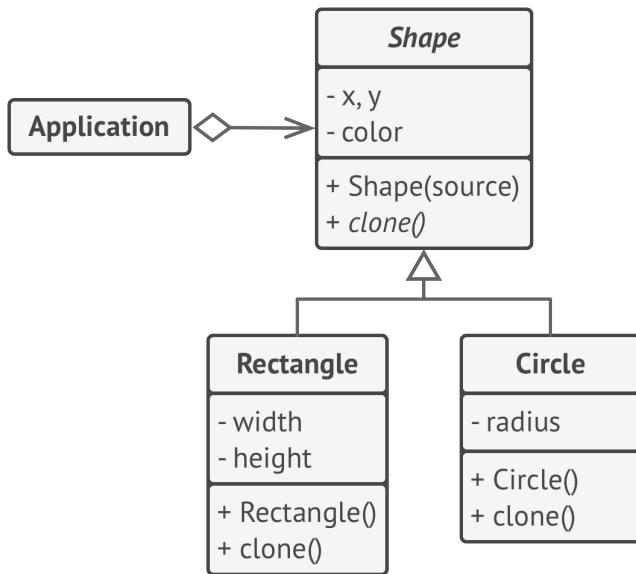


1. **프로토타입 레지스트리는** 자주 사용하는 프로토타입들에 쉽게 접근(액세스)하는 방법을 제공합니다. 이 레지스트리는 복사될 준비가 된 미리 만들어진 객체들의 집합을 저장합니다. 가장 간단한 프로토타입 레지스트리는 `name → prototype` 해시 맵입니다. 그러나 단순히 이름을 검색하는 것보다 더 나은 검색

기준이 필요한 경우 훨씬 더 탄탄한 레지스트리를 구축할 수 있습니다.

의사코드

아래 예시에서의 **프로토타입** 패턴은 코드를 기하학적 객체들의 클래스들에 결합하지 않고도 해당 객체들의 정확한 복사본을 생성할 수 있도록 합니다.



클래스 계층구조에 속한 객체 집합의 복제.

모든 **shape(모양)** 클래스는 같은 인터페이스를 따르며, 이 인터페이스는 복제 메서드를 제공합니다. 자식 클래스는 자신의 필드 값을 생성된 객체에 복사하기 전에 부모의 복제 메서드를 호출할 수 있습니다.

```

1 // 기초 프로토타입.
2 abstract class Shape is
3   field X: int
4   field Y: int
5   field color: string
6
7 // 일반 생성자.
8 constructor Shape() is
9   // ...
10
11 // 프로토타입 생성자. 기존 객체의 값들로 새로운 객체가 초기화됩니다.
12 constructor Shape(source: Shape) is
13   this()
14   this.X = source.X
15   this.Y = source.Y
16   this.color = source.color
17
18 // 복제 작업은 Shape(모양) 자식 클래스 중 하나를 반환합니다.
19 abstract method clone():Shape
20
21
22 // 구상 프로토타입. 복제 메서드는 현재 클래스의 생성자를 호출해 현재 객체를
23 // 생성자의 인수로 전달함으로써 한 번에 새로운 객체를 생성합니다. 생성자에서
24 // 실제로 모든 것을 복사하게 되면 결과의 일관성이 유지됩니다. 생성자가 새로운
25 // 객체가 완전히 완성되기 전까지 결과를 반환하지 않아서 어떤 객체도 일부분만 완성된
26 // 복제본을 참조할 수 없습니다.
27 class Rectangle extends Shape is
28   field width: int
29   field height: int
30
31 constructor Rectangle(source: Rectangle) is
32   // 부모 클래스에 정의된 비공개 필드들을 복사하려면 부모 생성자 호출이

```

```
33     // 필요합니다.
34     super(source)
35     this.width = source.width
36     this.height = source.height
37
38     method clone():Shape is
39         return new Rectangle(this)
40
41
42     class Circle extends Shape is
43         field radius: int
44
45         constructor Circle(source: Circle) is
46             super(source)
47             this.radius = source.radius
48
49         method clone():Shape is
50             return new Circle(this)
51
52
53     // 클라이언트 코드의 어딘가에...
54     class Application is
55         field shapes: array of Shape
56
57         constructor Application() is
58             Circle circle = new Circle()
59             circle.X = 10
60             circle.Y = 10
61             circle.radius = 20
62             shapes.add(circle)
63
64             Circle anotherCircle = circle.clone()
```

```

65     shapes.add(anotherCircle)
66     // `anotherCircle` 변수에는 `circle` 객체와 똑같은 사본이 포함되어
67     // 있습니다.
68
69     Rectangle rectangle = new Rectangle()
70     rectangle.width = 10
71     rectangle.height = 20
72     shapes.add(rectangle)
73
74 method businessLogic() is
75     // 프로토타입은 매우 유용합니다! 왜냐하면 프로토타입은 당신이 복사하려는
76     // 객체의 유형에 대해 아무것도 몰라도 복사본을 생성할 수 있도록 하기
77     // 때문입니다.
78     Array shapesCopy = new Array of Shapes.
79
80     // 예를 들어, 우리는 shapes(모양들) 배열의 정확한 요소들을 알지
81     // 못하며, 이 요소들이 모양이라는 것만 압니다. 그러나 다형성 덕분에
82     // 모양의 `clone`(복제) 메서드를 호출하면 프로그램이 모양의 실제
83     // 클래스를 확인하고 해당 클래스에 정의된 적절한 복제 메서드를
84     // 실행합니다. 그래서 우리가 단순한 모양 객체들의 집합이 아닌 적절한
85     // 복제본들을 얻는 것이죠.
86     foreach (s in shapes) do
87         shapesCopy.add(s.clone())
88
89     // `shapesCopy` (모양들의 복사본) 배열에는 `shape` (모양) 배열의
90     // 자식들과 정확히 일치하는 복사본들이 포함되어 있습니다.

```

💡 적용

💡 프로토타입 패턴은 복사해야 하는 객체들의 구상 클래스들에 코드가 의존하면 안 될 때 사용하세요.

- ⚡ 이와 같은 경우는 당신의 코드가 어떤 인터페이스를 통해 타사 코드에서 전달된 객체들과 함께 작동할 때 많이 발생합니다. 이러한 객체들의 구상 클래스들은 알려지지 않았기 때문에 이러한 클래스들에 의존할 수 없습니다.

프로토타입 패턴은 클라이언트 코드에 복제를 지원하는 모든 객체와 작업할 수 있도록 일반 인터페이스를 제공합니다. 이 인터페이스는 클라이언트 코드가 복제하는 객체들의 구상 클래스들에서 클라이언트 코드를 독립시킵니다.

- ⚡ **프로토타입 패턴은 각자의 객체를 초기화하는 방식만 다른 자식 클래스들의 수를 줄이고 싶을 때 사용하세요.**

- ⚡ 사용하기 전에 많은 설정이 필요한 복잡한 클래스가 있다고 가정해 봅시다. 이 클래스를 설정하는 데는 몇 가지 일반적인 방법들이 있으며 설정되어야 하는 클래스의 새로운 인스턴스들의 생성을 담당하는 코드는 당신의 앱에 흩어져 있습니다. 중복을 줄이기 위해 당신은 여러 자식 클래스들을 만들어 모든 공통 설정 코드를 그 클래스들의 생성자들에 넣었습니다. 이렇게 중복 문제는 해결했지만 이제 쓸모없는 자식 클래스들이 많이 생겼습니다.

프로토타입 패턴은 다양한 방식으로 설정된 미리 만들어진 객체들의 집합을 프로토타입들로 사용할 수 있도록 합니다. 일부 설정과 일치하는 자식 클래스를 인스턴스화하는 대신

클라이언트는 간단하게 적절한 프로토타입을 찾아 복제할 수 있습니다.

▣ 구현방법

1. 프로토타입 인터페이스를 생성한 후 그 안에 `clone` 메서드를 선언하세요. 또는 기존 계층 구조가 있는 경우, 이 메서드를 그 계층 구조의 모든 클래스들에 추가하세요.
2. 프로토타입 클래스는 이 클래스의 객체를 인수로 받아들이는 대체 생성자를 반드시 정의해야 합니다. 또 생성자는 이 클래스에 정의된 모든 필드의 값을 전달된 객체에서 새로 생성된 인스턴스로 복사해야 합니다. 또 자식 클래스를 변경할 때에는 부모 생성자를 호출하여 부모 클래스가 부모 클래스의 비공개 필드들의 복제를 처리하도록 해야 합니다.

현재 사용 중인 프로그래밍 언어가 메서드 오버로딩을 지원하지 않으면 별도의 '프로토타입' 생성자를 만들 수 없습니다. 따라서 객체의 데이터를 새로 생성된 복제본에 복사하는 작업은 `clone` (복제본) 메서드 내에서 수행되어야 합니다. 그래도 이 코드를 일반적인 생성자에 두는 것이 더 안전한 이유는 `new` 연산자를 호출한 직후에 생성된 객체는 완전히 설정된 상태로 반환되기 때문입니다.

3. 복제 메서드는 일반적으로 한 줄로 구성됩니다. 이 줄은 생성자의 프로토타입 버전으로 `new` 연산자를 실행합니다. 모든 클래스는 복제 메서드를 명시적으로 오버라이딩한 후 `new` 연산자와 함께 자체 클래스 이름을 사용해야 합니다. 그렇게 하지 않으면 복제 메서드가 부모 클래스의 객체를 생성할 수 있습니다.
4. 또, 추가 옵션으로 자주 사용하는 프로토타입들의 카탈로그를 저장할 중앙 프로토타입 레지스트리를 생성할 수 있습니다.

레지스트리를 새 팩토리 클래스로 구현하거나 레지스트리를 기초 프로토타입 클래스에 프로토타입을 가져오기 위한 정적 메서드와 함께 넣을 수 있습니다. 이 정적 메서드는 클라이언트 코드가 메서드에 전달하는 검색 기준을 기반으로 프로토타입을 검색해야 합니다. 이때 검색 기준은 단순한 문자열 태그이거나 복잡한 검색 매개변수들의 집합일 수 있습니다. 적절한 프로토타입을 찾고 나면, 레지스트리는 이를 복제한 후 복사본을 클라이언트에 반환해야 합니다.

마지막으로, 자식 클래스들의 생성자들에 대한 직접 호출들을 프로토타입 레지스트리의 팩토리 메서드에 대한 호출들로 대체하세요.

⚠️ 장단점

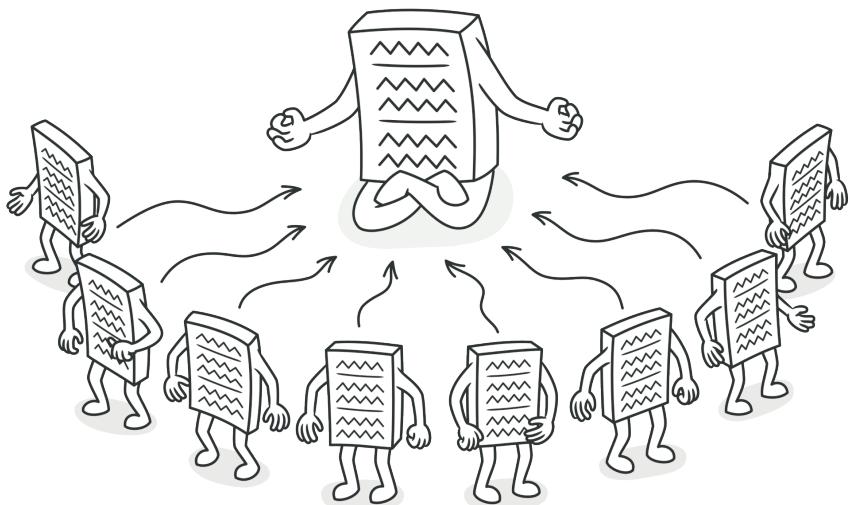
- ✓ 당신은 객체들을 그 구상 클래스들에 결합하지 않고 복제할 수 있습니다.

- ✓ 반복되는 초기화 코드를 제거한 후 그 대신 미리 만들어진 프로토타입들을 복제하는 방법을 사용할 수 있습니다.
- ✓ 복잡한 객체들을 더 쉽게 생성할 수 있습니다.
- ✓ 복잡한 객체들에 대한 사전 설정들을 처리할 때 상속 대신 사용할 수 있는 방법입니다.
- ✗ 순환 참조가 있는 복잡한 객체들을 복제하는 것은 매우 까다로울 수 있습니다.

↔ 다른 패턴과의 관계

- 많은 디자인은 복잡성이 낮고 자식 클래스들을 통해 더 많은 커스터마이징이 가능한 팩토리 메서드로 시작해 더 유연하면서도 더 복잡한 추상 팩토리, 프로토타입 또는 빌더 패턴으로 발전해 나갑니다.
- 추상 팩토리 클래스들은 팩토리 메서드들의 집합을 기반으로 하는 경우가 많습니다. 그러나 당신은 또한 프로토타입을 사용하여 추상 팩토리의 구상 클래스들의 생성 메서드들을 구현할 수도 있습니다.
- 프로토타입은 커맨드 패턴의 복사본들을 기록에 저장해야 할 때 도움이 될 수 있습니다.

- **데코레이터** 및 **복합체** 패턴을 많이 사용하는 디자인들은 **프로토타입**을 사용하면 종종 이득을 볼 수 있습니다. 프로토타입 패턴을 적용하면 복잡한 구조들을 처음부터 다시 건축하는 대신 복제할 수 있기 때문입니다.
- **프로토타입**은 상속을 기반으로 하지 않으므로 상속과 관련된 단점들이 없습니다. 반면에 **프로토타입**은 복제된 객체의 복잡한 초기화가 필요합니다. **팩토리 메서드**는 상속을 기반으로 하지만 초기화 단계가 필요하지 않습니다.
- 때로는 **프로토타입이 메멘토** 패턴의 더 간단한 대안이 될 수 있으며, 이 패턴은 상태를 기록에 저장하려는 객체가 간단하고 외부 리소스에 대한 링크가 없거나 링크들이 있어도 이들을 재설정하기 쉬운 경우에 작동합니다.
- **추상 팩토리들**, **빌더들** 및 **프로토타입들은** 모두 **싱글턴**으로 구현할 수 있습니다.



싱글턴 패턴

다음 이름으로도 불립니다: Singleton

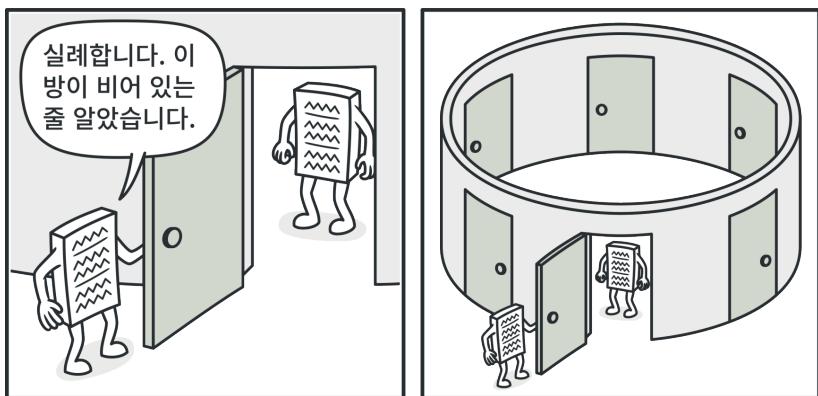
싱글턴은 클래스에 인스턴스가 하나만 있도록 하면서 이 인스턴스에 대한 전역 접근(액세스) 지점을 제공하는 생성 디자인 패턴입니다.

(:(문제

싱글턴 패턴은 한 번에 두 가지의 문제를 동시에 해결함으로써 단일 책임 원칙을 위반합니다.

- 클래스에 인스턴스가 하나만 있도록 합니다.** 사람들은 클래스에 있는 인스턴스 수를 제어하려는 가장 일반적인 이유는 일부 공유 리소스(예: 데이터베이스 또는 파일)에 대한 접근을 제어하기 위함입니다.

예를 들어 객체를 생성했지만 잠시 후 새 객체를 생성하기로 했다고 가정해 봅시다. 그러면 새 객체를 생성하는 대신 이미 만든 객체를 받게 됩니다. 물론 생성자 호출은 특성상 반드시 새 객체를 반환해야 하므로 위 행동은 일반 생성자로 구현할 수 없습니다.



클라이언트들은 항상 같은 객체와 작업하고 있다는 사실을 인식조차 못 할 수 있습니다.

2. 해당 인스턴스에 대한 전역 접근 지점을 제공합니다. 필수 객체들을 저장하기 위해 전역 변수들을 정의했다고 가정해 봅시다. 이 변수들을 사용하면 매우 편리할지는 몰라도, 모든 코드가 잠재적으로 해당 변수의 내용을 덮어쓸 수 있고 그로 인해 앱에 오류가 발생해 충돌할 수 있으므로 그리 안전한 방법은 아닙니다.

전역 변수와 마찬가지로 싱글턴 패턴을 사용하면 프로그램의 모든 곳에서부터 일부 객체에 접근할 수 있습니다. 그러나 이 패턴은 다른 코드가 해당 인스턴스를 덮어쓰지 못하도록 보호하기도 합니다.

이 문제에는 또 다른 측면이 있습니다. 당신은 첫 번째 문제를 해결하는 코드가 프로그램 전체에 흩어져 있는 것을 원하지 않을 것입니다. 특히 코드의 나머지 부분이 이미 첫 번째 문제를 해결하는 코드에 의존하고 있다면, 이 코드를 한 클래스 내에 두는 것이 훨씬 좋습니다.

최근에는 싱글턴 패턴이 워낙 대중화되어 패턴이 나열된 문제 중 한 가지만 해결하더라도 그것을 싱글턴이라고 부를 수 있습니다.

😊 해결책

싱글턴의 모든 구현은 공통적으로 다음의 두 단계를 갖습니다.

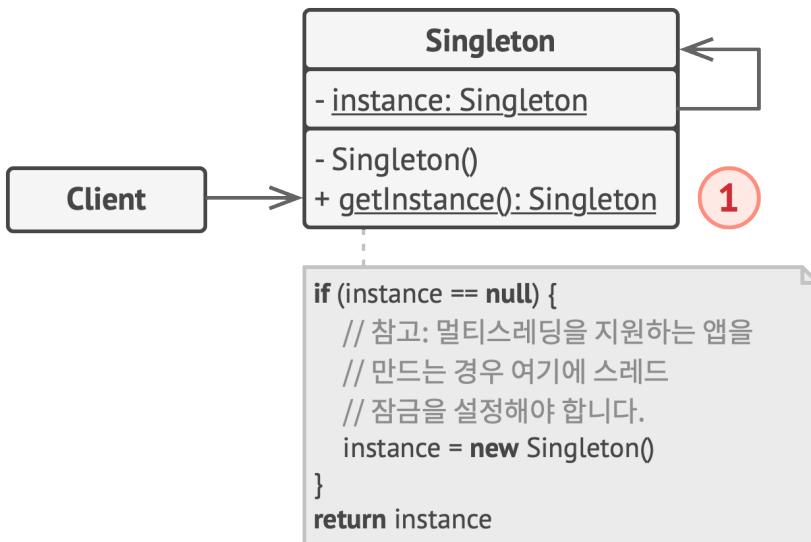
- 다른 객체들이 싱글턴 클래스와 함께 `new` 연산자를 사용하지 못하도록 디폴트 생성자를 비공개로 설정하세요.
- 생성자 역할을 하는 정적 생성 메서드를 만드세요. 내부적으로 이 메서드는 객체를 만들기 위하여 비공개 생성자를 호출한 후 객체를 정적 필드에 저장합니다. 이 메서드에 대한 그다음 호출들은 모두 캐시된 객체를 반환합니다.

당신의 코드가 싱글턴 클래스에 접근할 수 있는 경우, 이 코드는 싱글턴의 정적 메서드를 호출할 수 있습니다. 따라서 해당 메서드가 호출될 때마다 항상 같은 객체가 반환됩니다.

🚗 실제상황 적용

정부는 싱글턴 패턴의 훌륭한 예입니다. 국가는 하나의 공식 정부만 가질 수 있습니다. 그리고 'X의 정부'라는 명칭은 정부를 구성하는 개인들의 신원과 관계없이 정부 책임자들의 그룹을 식별하는 글로벌 접근 지점입니다.

구조



1. **싱글턴** 클래스는 정적 메서드 `getInstance`를 선언합니다. 이 메서드는 자체 클래스의 같은 인스턴스를 반환합니다.

싱글턴의 생성자는 항상 클라이언트 코드에서부터 숨겨져야 합니다. `getInstance` 메서드를 호출하는 것이 Singleton 객체를 가져올 수 있는 유일한 방법이어야 합니다.

의사코드

이 예에서 데이터베이스의 연결 클래스는 **싱글턴**의 역할을 합니다. 이 클래스에는 공개된 생성자가 없으므로 해당 클래스의 객체를 가져오는 유일한 방법은 `getInstance` 메서드를 호출하는

것입니다. 이 메서드는 처음 생성된 객체를 캐시 한 후 모든 후속 호출들에서 해당 객체를 반환합니다.

```

1 // 데이터베이스 클래스는 클라이언트들이 프로그램 전체에서 데이터베이스 연결의 같은
2 // 인스턴스에 접근할 수 있도록 해주는 `getInstance` (인스턴스 가져오기) 메서드를
3 // 정의합니다.
4 class Database is
5     // 싱글턴 인스턴스를 저장하기 위한 필드는 정적으로 선언되어야 합니다.
6     private static field instance: Database
7
8     // 싱글턴의 생성자는 `new` 연산자를 사용한 직접 생성 호출들을 방지하기 위해
9     // 항상 비공개여야 합니다.
10    private constructor Database() is
11        // 데이터베이스 서버에 대한 실제 연결과 같은 일부 초기화 코드.
12
13    // 싱글턴 인스턴스로의 접근을 제어하는 정적 메서드.
14    public static method getInstance() is
15        if (Database.instance == null) then
16            acquireThreadLock() and then
17                // 이 스레드가 잠금 해제를 기다리는 동안 인스턴스가 다른
18                // 스레드에 의해 초기화되지 않았는지 확인하세요.
19                if (Database.instance == null) then
20                    Database.instance = new Database()
21
22        return Database.instance
23
24    // 마지막으로 모든 싱글턴은 해당 로직의 인스턴스에서 실행할 수 있는 비즈니스
25    // 로직을 정의해야 합니다.
26    public method query(sql) is
27        // 예를 들어 앱의 모든 데이터베이스 쿼리들은 이 메서드를 거칩니다. 따라서
28        // 여기에 스로틀링 또는 캐싱 논리를 배치할 수 있습니다.
29        // ...

```

```

29
30 class Application is
31 method main() is
32     Database foo = Database.getInstance()
33     foo.query("SELECT ...")
34     // ...
35     Database bar = Database.getInstance()
36     bar.query("SELECT ...")
37 // 변수 `bar`는 변수 `foo`와 같은 객체를 포함할 것입니다.

```

💡 적용

- 💡 싱글턴 패턴은 당신 프로그램의 클래스에 모든 클라이언트가 사용할 수 있는 단일 인스턴스만 있어야 할 때 사용하세요. 예를 들자면 프로그램의 다른 부분들에서 공유되는 단일 데이터베이스 객체처럼 말입니다.
- 💡 싱글턴 패턴은 특별 생성 메서드를 제외하고는 클래스의 객체들을 생성할 수 있는 모든 다른 수단들을 비활성화합니다. 이 메서드는 새 객체를 생성하거나 객체가 이미 생성되었으면 기존 객체를 반환합니다.
- 💡 싱글턴 패턴은 전역 변수들을 더 엄격하게 제어해야 할 때 사용하세요.

- ▶ 전역 변수들과 달리 싱글턴 패턴은 클래스의 인스턴스가 하나만 있도록 보장해 줍니다. 캐시 된 인스턴스는 싱글턴 클래스 자체를 제외하고는 그 어떤 것과도 대체될 수 없습니다.

참고로 이 제한은 언제든 조정할 수 있고 원하는 수만큼의 싱글턴 인스턴스 생성을 허용할 수 있습니다. 그러기 위해서 변경해야 하는 코드의 유일한 부분은 `getInstance` 메서드의 본문입니다.

▣ 구현방법

1. 싱글턴 인스턴스의 저장을 위해 클래스에 비공개 정적 필드를 추가하세요.
2. 싱글턴 인스턴스를 가져오기 위한 공개된 정적 생성 메서드를 선언하세요.
3. 정적 메서드 내에서 '지연된 초기화'를 구현하세요. 그러면 이것은 첫 번째 호출에서 새 객체를 만든 후 그 객체를 정적 필드에 넣을 것입니다. 이 메서드는 모든 후속 호출들에서 항상 해당 인스턴스를 반환해야 합니다.
4. 클래스의 생성자를 비공개로 만드세요. 그러면 클래스의 정적 메서드는 여전히 생성자를 호출할 수 있지만 다른 객체들은 호출할 수 없을 것입니다.

5. 클라이언트 코드를 살펴보며 싱글턴의 생성자에 대한 모든 직접 호출들을 싱글턴의 정적 생성 메서드에 대한 호출로 바꾸세요.

⚠️ 장단점

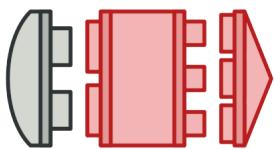
- ✓ 클래스가 하나의 인스턴트만 갖는다는 것을 확신할 수 있습니다.
- ✓ 이 인스턴스에 대한 전역 접근 지점을 얻습니다.
- ✓ 싱글턴 객체는 처음 요청될 때만 초기화됩니다.
- ✗ 단일 책임 원칙을 위반합니다. 이 패턴은 한 번에 두 가지의 문제를 동시에 해결합니다.
- ✗ 또 싱글턴 패턴은 잘못된 디자인(예를 들어 프로그램의 컴포넌트들이 서로에 대해 너무 많이 알고 있는 경우)을 가릴 수 있습니다.
- ✗ 그리고 이 패턴은 다중 스레드 환경에서 여러 스레드가 싱글턴 객체를 여러 번 생성하지 않도록 특별한 처리가 필요합니다.
- ✗ 싱글턴의 클라이언트 코드를 유닛 테스트하기 어려울 수 있습니다. 그 이유는 많은 테스트 프레임워크들이 모의 객체들을 생성할 때 상속에 의존하기 때문입니다. 싱글턴 클래스의 생성자는 비공개이고 대부분 언어에서 정적 메서드를 오버라이딩하는 것이 불가능하므로 싱글턴의 한계를 극복할 수 있는 창의적인 방법을 생각해야 합니다. 아니면 그냥 테스트를 작성하지 말거나 싱글턴 패턴을 사용하지 않으면 됩니다.

↔ 다른 패턴과의 관계

- 대부분의 경우 하나의 퍼사드 객체만 있어도 충분하므로 퍼사드 패턴의 클래스는 종종 싱글턴으로 변환될 수 있습니다.
- 만약 객체들의 공유된 상태들을 단 하나의 플라이웨이트 객체로 줄일 수 있다면 플라이웨이트는 싱글턴과 유사해질 수 있습니다. 그러나 이 패턴들에는 두 가지 근본적인 차이점이 있습니다:
 1. 싱글턴은 인스턴스가 하나만 있어야 합니다. 반면에 플라이웨이트 클래스는 여러 고유한 상태를 가진 여러 인스턴스를 포함할 수 있습니다.
 2. 싱글턴 객체는 변할 수 있습니다 (mutable). 플라이웨이트 객체들은 변할 수 없습니다 (immutable).
- 추상 팩토리들, 빌더들 및 프로토타입들은 모두 싱글턴으로 구현할 수 있습니다.

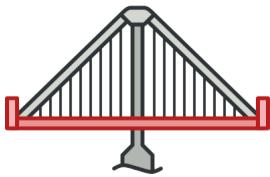
구조 패턴

구조 패턴은 구조를 유연하고 효율적으로 유지하면서 객체들과 클래스들을 더 큰 구조로 조립하는 방법을 설명합니다.



어댑터

호환되지 않는 인터페이스를 가진 객체들이 협업할 수 있도록 합니다.



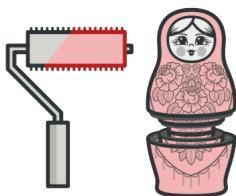
브리지

큰 클래스 또는 밀접하게 관련된 클래스들의 집합을 두 개의 개별 계층구조(추상화 및 구현)로 나눈 후 각각 독립적으로 개발할 수 있도록 합니다.



복합체

객체들을 트리 구조들로 구성한 후, 이러한 트리 구조들이 개별 객체들인 것처럼 작업할 수 있도록 하는 디자인 패턴입니다



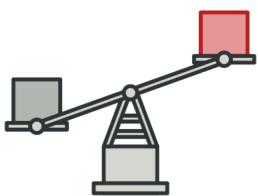
데코레이터

객체들을 새로운 행동들을 포함한 특수 래퍼 객체들 내에 넣어서 위 행동들을 해당 객체들에 연결시킵니다.



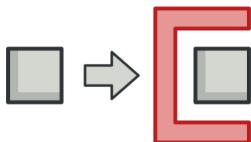
퍼사드

라이브러리에 대한, 프레임워크에 대한 또는 다른 클래스들의 복잡한 집합에 대한 단순화된 인터페이스를 제공합니다.



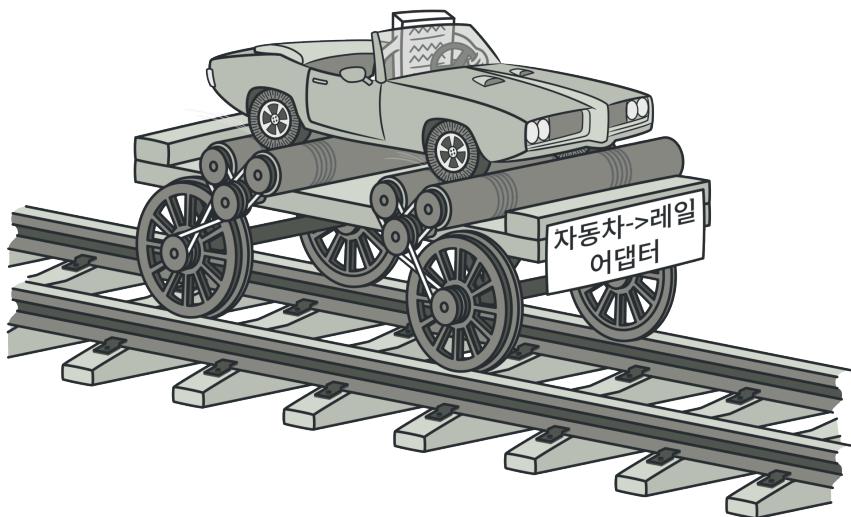
플라이웨이트

각 객체에 모든 데이터를 유지하는 대신 여러 객체 간에 상태의 공통 부분들을 공유하여 사용할 수 있는 RAM에 더 많은 객체를 포함할 수 있도록 합니다.



프록시

다른 객체에 대한 대체 또는 자리표시자를 제공할 수 있습니다.
프록시는 원래 객체에 대한 접근을 제어하므로, 당신의 요청이 원래
객체에 전달되기 전 또는 후에 무언가를 수행할 수 있도록 합니다.



어댑터 패턴

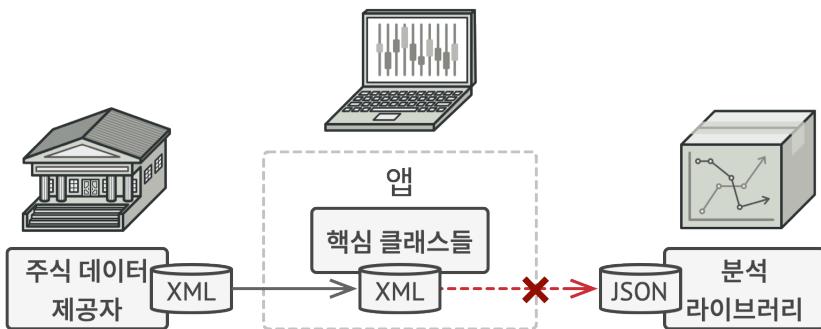
다음 이름으로도 불립니다: 래퍼(Wrapper), Adapter

어댑터는 호환되지 않는 인터페이스를 가진 객체들이 협업할 수 있도록 하는 구조적 디자인 패턴입니다.

(:) 문제

주식 시장 모니터링 앱을 만들고 있고, 이 앱은 여러 소스에서 주식 데이터를 XML 형식으로 다운로드한 후 사용자에게 보기 좋은 차트들과 다이어그램들을 표시한다고 상상해 봅시다.

어느 시점에 당신은 타사의 스마트 분석 라이브러리를 통합하여 당신의 앱을 개선하기로 결정했습니다. 그런데 함정이 있습니다: 이 분석 라이브러리는 JSON 형식의 데이터로만 작동한다는 것입니다.



위 분석 라이브러리는 '있는 그대로' 사용할 수 없습니다. 왜냐하면 당신의 앱과 호환되지 않는 형식의 데이터를 기다리고 있기 때문입니다.

당신은 이 라이브러리를 XML과 작동하도록 변경하도록 수 있으나, 그러면 라이브러리에 의존하는 일부 기존 코드가 손상될 수 있습니다. 또 처음부터 타사의 라이브러리 소스 코드에 접근하는 것이 불가능하여 위의 해결 방식을 사용하지 못할 수도 있습니다.

😊 해결책

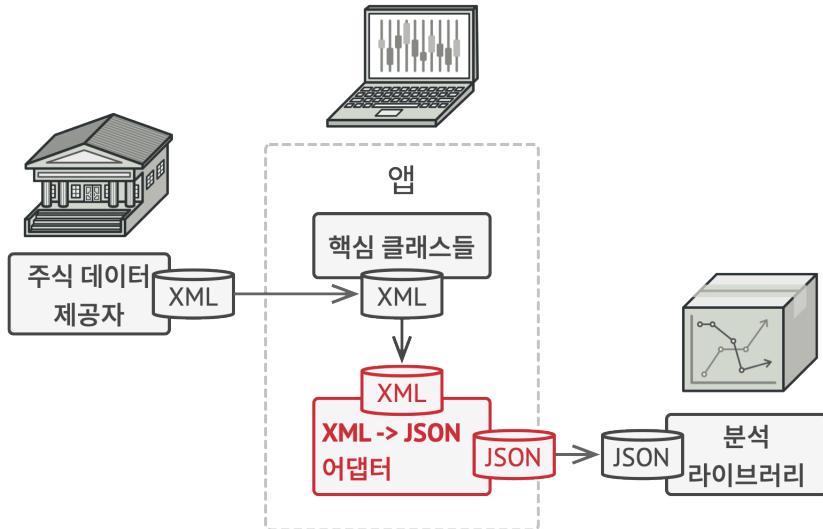
당신은 어댑터를 만들 수 있습니다. 어댑터는 한 객체의 인터페이스를 다른 객체가 이해할 수 있도록 변환하는 특별한 객체입니다.

어댑터는 변환의 복잡성을 숨기기 위하여 객체 중 하나를 래핑(포장)합니다. 래핑된 객체는 어댑터를 인식하지도 못합니다. 예를 들어 미터 및 킬로미터 단위로 작동하는 객체를 모든 데이터를 피트 및 마일과 같은 영국식 단위로 변환하는 어댑터로 래핑할 수 있습니다.

어댑터는 데이터를 다양한 형식으로 변환할 수 있을 뿐만 아니라 다른 인터페이스를 가진 객체들이 협업하는 데에도 도움을 줄 수 있으며, 대략 다음과 같이 작동합니다:

1. 어댑터는 기존에 있던 객체 중 하나와 호환되는 인터페이스를 받습니다.
2. 이 인터페이스를 사용하면 기존 객체는 어댑터의 메서드들을 안전하게 호출할 수 있습니다.
3. 호출을 수신하면 어댑터는 이 요청을 두 번째 객체에 해당 객체가 예상하는 형식과 순서대로 전달합니다.

때로는 양방향으로 호출을 변환할 수 있는 양방향 어댑터를 만드는 것도 가능합니다.



다시 당신의 주식 시장 앱을 살펴봅시다. 당신은 형식이 호환되지 않는 문제를 해결하기 위해 당신의 코드와 직접 작동하는 분석 라이브러리의 모든 클래스에 대한 XML->JSON 변환 어댑터를 만듭니다. 그 후 이러한 어댑터들을 통해서만 해당 라이브러리와 통신하도록 코드를 조정합니다. 어댑터는 호출을 받으면 들어오는 XML 데이터를 JSON 구조로 변환한 후 해당 호출을 래핑된 분석 객체의 적절한 메서드들에 전달합니다.

🚗 실제상황 적용

해외로 여행갈 때



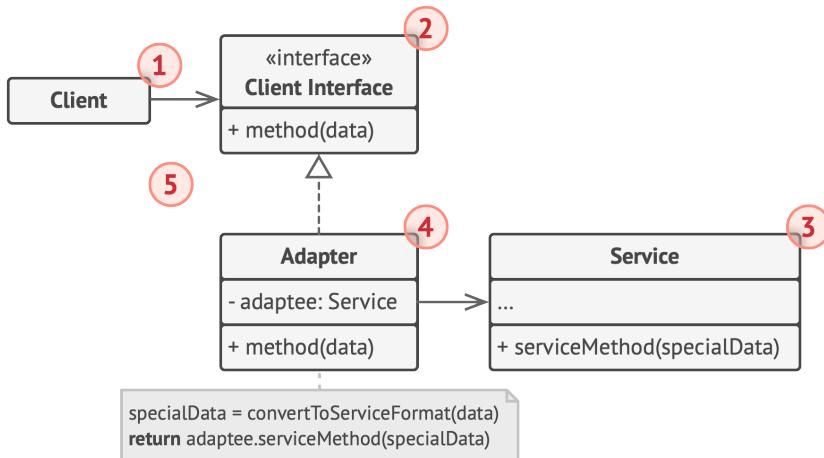
해외여행 전과 후의 서류가방.

미국에서 유럽으로 처음 여행을 가서 노트북을 충전하려고 하면 깜짝 놀랄지도 모릅니다. 전원 플러그와 소켓은 국가마다 표준이 달라 미국 플러그가 독일 소켓에 맞지 않을 수 있기 때문입니다. 이 문제는 미국식 소켓과 유럽식 플러그가 있는 전원 플러그 어댑터를 사용하면 해결할 수 있습니다.

구조

객체 어댑터

이 구현은 객체 합성 원칙을 사용합니다. 어댑터는 한 객체의 인터페이스를 구현하고 다른 객체는 래핑합니다. 위 합성은 모든 인기 있는 프로그래밍 언어로 구현할 수 있습니다.



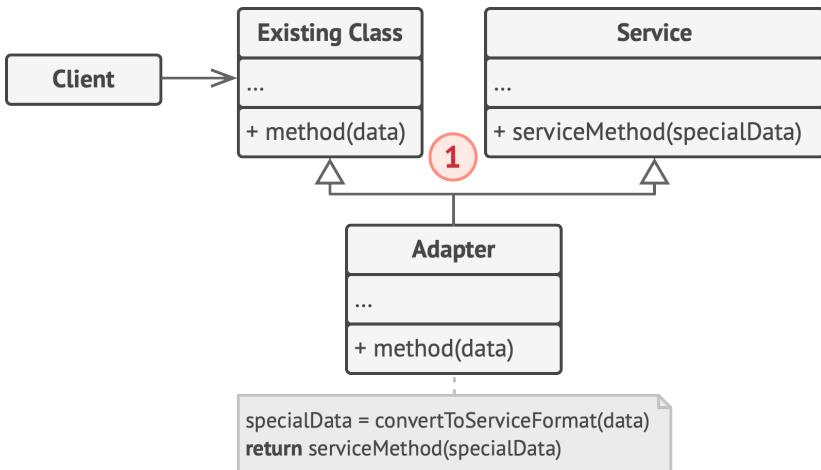
1. **클라이언트**는 프로그램의 기존 비즈니스 로직을 포함하는 클래스입니다.
2. **클라이언트 인터페이스**는 다른 클래스들이 클라이언트 코드와 공동 작업할 수 있도록 따라야 하는 프로토콜을 뜻합니다.
3. **서비스**는 일반적으로 타사 또는 레거시의 유용한 클래스를 뜻합니다. 클라이언트는 서비스 클래스를 직접 사용할 수

없습니다. 왜냐하면 서비스 클래스는 호환되지 않는 인터페이스를 가지고 있기 때문입니다.

4. **어댑터**는 클라이언트와 서비스 양쪽에서 작동할 수 있는 클래스로, 서비스 객체를 래핑하는 동안 클라이언트 인터페이스를 구현합니다. 어댑터는 어댑터 인터페이스를 통해 클라이언트로부터 호출들을 수신한 후 이 호출을 래핑된 서비스 객체가 이해할 수 있는 형식의 호출들로 변환합니다.
5. 클라이언트 코드는 클라이언트 인터페이스를 통해 어댑터와 작동하는 한 구상 어댑터 클래스와 결합하지 않습니다. 덕분에 기존 클라이언트 코드를 손상하지 않고 새로운 유형의 어댑터들을 프로그램에 도입할 수 있습니다. 이것은 서비스 클래스의 인터페이스가 변경되거나 교체될 때 유용할 수 있습니다: 클라이언트 코드를 변경하지 않은 채 새 어댑터 클래스를 생성할 수 있으니까요.

클래스 어댑터

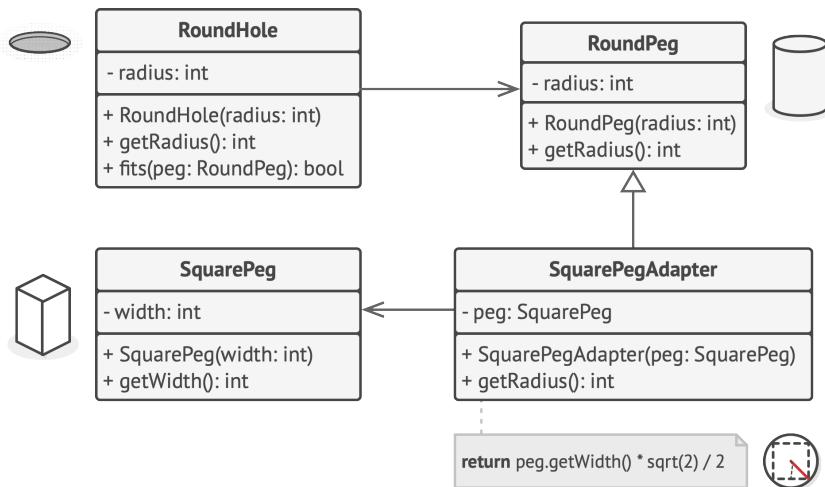
이 구현은 상속을 사용하며, 어댑터는 동시에 두 객체의 인터페이스를 상속합니다. 이 방식은 C++ 와 같이 다중 상속을 지원하는 프로그래밍 언어에서만 구현할 수 있습니다.



1. **클래스 어댑터**는 객체를 래핑할 필요가 없습니다. 그 이유는 클라이언트와 서비스 양쪽에서 행동들을 상속받기 때문입니다. 위의 어댑테이션(적용)은 오버라이딩된 메서드 내에서 발생합니다. 위 어댑터는 기존 클라이언트 클래스 대신 사용할 수 있습니다.

의사코드

이 **어댑터** 패턴은 서로 맞지 않는 정사각형 못과 둥근 구멍이라는 고전적인 예시를 기초로 합니다.



둥근 구멍에 정사각형 못을 맞춰 넣기.

어댑터는 정사각형 지름의 절반(즉, 사각형 못을 수용할 수 있는 가장 작은 원의 반지름)을 반지름으로 가진 둥근 못인 척 합니다.

```

1 // RoundHole(둥근 구멍) 및 RoundPeg(둥근 못)라는 호환되는 인터페이스들이 있는
2 // 두 개의 클래스가 있다고 가정해봅시다.
3 class RoundHole is
4   constructor RoundHole(radius) { ... }
5
6   method getRadius() is
  
```

```
7      // 구멍의 반지름을 반환하세요.
8
9  method fits(peg: RoundPeg) is
10    return this.getRadius() >= peg.getRadius()
11
12 class RoundPeg is
13   constructor RoundPeg(radius) { ... }
14
15   method getRadius() is
16     // 뭇의 반지름을 반환하세요.
17
18
19 // 그러나 SquarePeg(직사각형 뭇)라는 호환되지 않는 클래스가 있습니다.
20 class SquarePeg is
21   constructor SquarePeg(width) { ... }
22
23   method getWidth() is
24     // 직사각형 뭇의 너비를 반환하세요.
25
26
27 // 어댑터 클래스를 사용하면 정사각형 뭇을 둥근 구멍에 맞출 수 있습니다. 어댑터
28 // 객체들은 RoundPeg(둥근 뭇) 클래스를 확장해 둥근 뭇들처럼 작동하게 해줍니다.
29 class SquarePegAdapter extends RoundPeg is
30   // 실제로 어댑터에는 SquarePeg(정사각형 뭇) 클래스의 인스턴스가 포함되어
31   // 있습니다.
32   private field peg: SquarePeg
33
34   constructor SquarePegAdapter(peg: SquarePeg) is
35     this.peg = peg
36
37   method getRadius() is
38     // 어댑터는 이것이 어댑터가 실제로 감싸는 정사각형 뭇에 맞는 반지름을
```

```

39     // 가진 원형 못인 것처럼 가장합니다.
40     return peg.getWidth() * Math.sqrt(2) / 2
41
42
43 // 클라이언트 코드 어딘가에...
44 hole = new RoundHole(5)
45 rpeg = new RoundPeg(5)
46 hole.fits(rpeg) // 참
47
48 small_sqpeg = new SquarePeg(5)
49 large_sqpeg = new SquarePeg(10)
50 hole.fits(small_sqpeg) // 이것은 컴파일되지 않습니다(호환되지 않는 유형)
51
52 small_sqpeg_adapter = new SquarePegAdapter(small_sqpeg)
53 large_sqpeg_adapter = new SquarePegAdapter(large_sqpeg)
54 hole.fits(small_sqpeg_adapter) // 참
55 hole.fits(large_sqpeg_adapter) // 거짓

```

💡 적용

💡 어댑터 클래스는 기존 클래스를 사용하고 싶지만 그 인터페이스가 나머지 코드와 호환되지 않을 때 사용하세요.

⚡️ 어댑터 패턴은 당신의 코드와 레거시 클래스, 타사 클래스 또는 특이한 인터페이스가 있는 다른 클래스 간의 변환기 역할을 하는 중간 레이어 클래스를 만들 수 있도록 합니다.

 이 패턴은 부모 클래스에 추가할 수 없는 어떤 공통 기능들이 없는 여러 기존 자식 클래스들을 재사용하려는 경우에 사용하세요.

 각 자식 클래스를 확장한 후 누락된 기능들을 새 자식 클래스들에 넣을 수 있습니다. 하지만 해당 코드를 모든 새 클래스들에 복제해야 하며, 그건 정말 나쁜 냄새가 나는 코드일 것입니다.

이보다 훨씬 더 깔끔한 해결책은 누락된 기능을 어댑터 클래스에 넣는 것입니다. 그 후 어댑터 내부에 누락된 기능이 있는 객체들을 래핑하면 필요한 기능들을 동적으로 얻을 것입니다. 이 해결책이 작동하려면 대상 클래스들에는 반드시 공통 인터페이스가 있어야 하며 어댑터의 필드는 해당 인터페이스를 따라야 합니다. 위 접근 방식은 데코레이터 패턴과 매우 유사합니다.

구현방법

- 호환되지 않는 인터페이스가 있는 클래스가 최소 두 개 이상 있는지 확인하세요:

- 당신이 변경할 수 없는 유용한 서비스 클래스가 있습니다. (종종 타사 코드, 레거시 코드 또는 기존 의존성이 많은 코드).
- 위 서비스 클래스를 사용하여 이득을 얻을 수 있는 하나 또는 여러 개의 클라이언트 클래스들이 있습니다.

2. 클라이언트 인터페이스를 선언하고 클라이언트들이 서비스와 통신하는 방법을 기술하세요.
3. 어댑터 클래스를 생성한 후 클라이언트 인터페이스를 따르게 하세요. 일단은 모든 메서드들을 비워 두세요.
4. 서비스 객체에 참조를 저장하기 위하여 어댑터 클래스에 필드를 추가하세요. 일반적으로 사용되는 방법은 생성자를 통해 이 필드를 초기화하는 것이지만, 때때로 어댑터의 메서드들을 호출할 때는 이 필드를 어댑터에 전달하는 것이 더 편리하기도 합니다.
5. 클라이언트 인터페이스의 모든 메서드를 어댑터 클래스에서 하나씩 구현하세요. 어댑터는 인터페이스 또는 데이터 형식 변환만 처리해야 하며, 실제 작업의 대부분을 서비스 객체에 위임해야 합니다.
6. 클라이언트들은 클라이언트 인터페이스를 통해 어댑터를 사용해야 합니다. 이렇게 하면 클라이언트 코드에 영향을 주지 않고 어댑터들을 변경하거나 확장할 수 있습니다.

⚠️ 장단점

- ✓ 단일 책임 원칙. 프로그램의 기본 비즈니스 로직에서 인터페이스 또는 데이터 변환 코드를 분리할 수 있습니다.
- ✓ 개방/폐쇄 원칙. 클라이언트 코드가 클라이언트 인터페이스를 통해 어댑터와 작동하는 한, 기존의 클라이언트 코드를

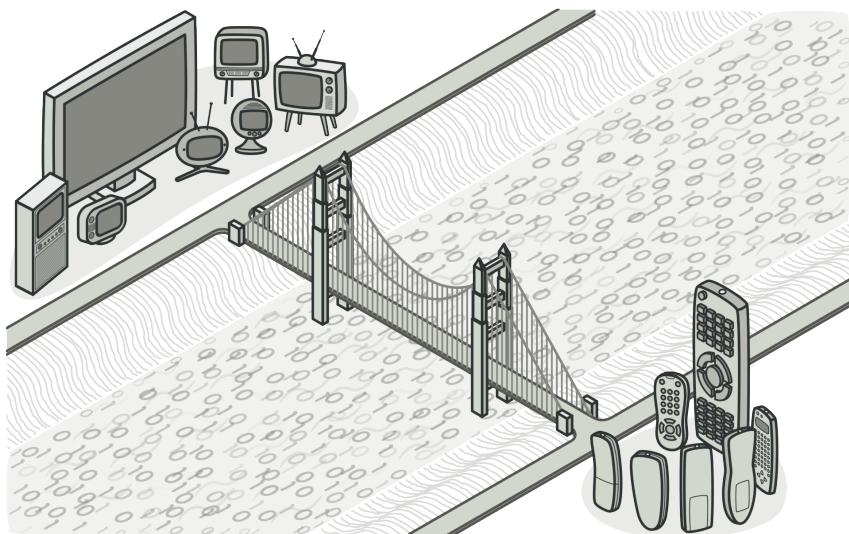
손상시키지 않고 새로운 유형의 어댑터들을 프로그램에 도입할 수 있습니다.

- ✖ 다수의 새로운 인터페이스와 클래스들을 도입해야 하므로 코드의 전반적인 복잡성이 증가합니다. 때로는 코드의 나머지 부분과 작동하도록 서비스 클래스를 변경하는 것이 더 간단합니다.

↔ 다른 패턴과의 관계

- 브리지는 일반적으로 사전에 설계되며, 앱의 다양한 부분을 독립적으로 개발할 수 있도록 합니다. 반면에 어댑터는 일반적으로 기존 앱과 사용되어 원래 호환되지 않던 일부 클래스들이 서로 잘 작동하도록 합니다.
- 어댑터는 기존 객체의 인터페이스를 변경하는 반면 데코레이터는 객체를 해당 객체의 인터페이스를 변경하지 않고 향상합니다. 또한 데코레이터는 어댑터를 사용할 때는 불가능한 재귀적 합성을 지원합니다.
- 어댑터는 다른 인터페이스를, 프록시는 같은 인터페이스를, 데코레이터는 향상된 인터페이스를 래핑된 객체에 제공합니다.
- 퍼사드는 기존 객체들을 위한 새 인터페이스를 정의하는 반면 어댑터는 기존의 인터페이스를 사용할 수 있게 만들려고 노력합니다. 또 어댑터는 일반적으로 하나의 객체만 래핑하는 반면 퍼사드는 많은 객체의 하위시스템과 함께 작동합니다.

- **브리지, 상태, 전략** 패턴은 매우 유사한 구조로 되어 있으며, **어댑터** 패턴도 이들과 어느 정도 유사한 구조로 되어 있습니다. 위 모든 패턴은 다른 객체에 작업을 위임하는 합성을 기반으로 합니다. 하지만 이 패턴들은 모두 다른 문제들을 해결합니다. 패턴은 특정 방식으로 코드의 구조를 짜는 레시피에 불과하지 않습니다. 왜냐하면 패턴은 해결하는 문제를 다른 개발자들에게 전달할 수도 있기 때문입니다.



브리지 패턴

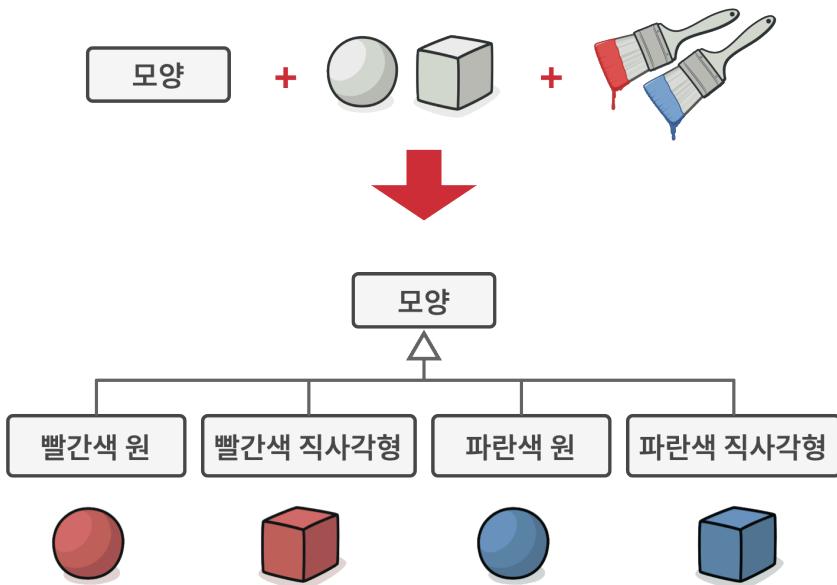
다음 이름으로도 불립니다: Bridge

브리지는 큰 클래스 또는 밀접하게 관련된 클래스들의 집합을 두 개의 개별 계층구조(추상화 및 구현)로 나눈 후 각각 독립적으로 개발할 수 있도록 하는 구조 디자인 패턴입니다.

:(문제

추상화? 구현? 어렵게 들리시나요? 진정하세요. 그리고 간단한 예시를 한번 살펴봅시다.

`Circle` (원) 및 `Square` (직사각형)라는 한 쌍의 자식 클래스들이 있는 기하학적 `Shape` (모양) 클래스가 있다고 가정해 봅시다. 이 클래스 계층 구조를 확장하여 색상을 도입하기 위해 `Red` (빨간색) 및 `Blue` (파란색) 모양들의 자식 클래스들을 만들 계획입니다. 그러나 이미 두 개의 자식 클래스가 있으므로 `Blue-Circle` (파란색 원) 및 `RedSquare` (빨간색 직사각형)와 같은 네 가지의 클래스 조합을 만들어야 합니다.



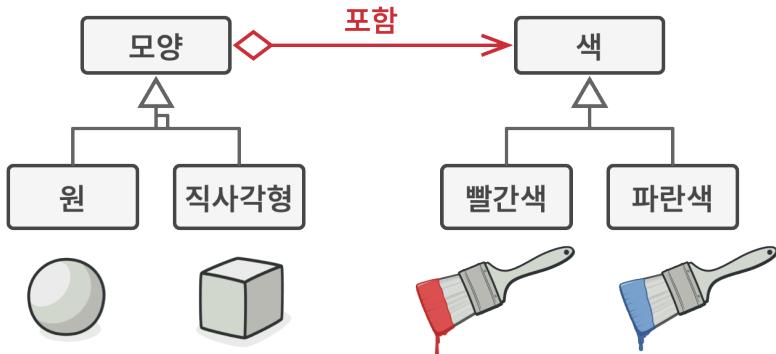
클래스 조합들의 수는 기하급수적으로 증가합니다.

새로운 모양 유형들과 색상 유형들을 추가할 때마다 계층 구조는 기하급수적으로 성장합니다. 예를 들어, 삼각형 모양을 추가하려면 각 색상별로 하나씩 두 개의 자식 클래스들을 도입해야 합니다. 그리고 그 후에 또 새 색상을 추가하려면 각 모양 유형별로 하나씩 세 개의 자식 클래스를 만들어야 합니다. 유형들이 많아지면 많아질수록 코드는 점점 복잡해집니다.

😊 해결책

이 문제는 모양과 색상의 두 가지 독립적인 차원에서 모양 클래스들을 확장하려고 하기 때문에 발생합니다. 이것은 클래스 상속과 관련된 매우 일반적인 문제입니다.

브리지 패턴은 상속에서 객체 합성으로 전환하여 이 문제를 해결하려고 시도합니다. 이것이 의미하는 바는 차원 중 하나를 별도의 클래스 계층구조로 추출하여 원래 클래스들이 한 클래스 내에서 모든 상태와 행동들을 갖는 대신 새 계층구조의 객체를 참조하도록 한다는 것입니다.



클래스 계층구조의 기하급수적인 성장을 방지하기 위하여 그것을 여러 관련 계층구조들로 변환할 수 있습니다.

이 접근 방식을 따르면, 색상 관련 코드를 Red 및 Blue라는 두 개의 자식 클래스들이 있는 자체 클래스로 추출할 수 있습니다. 그런 다음 Shape 클래스는 색상 객체들 중 하나를 가리키는 참조 필드를 받습니다. 이제 모양은 연결된 색상 객체에 모든 색상 관련 작업을 위임할 수 있습니다. 이 참조는 Shape 및 Color 클래스들 사이의 브리지(다리) 역할을 할 것입니다. 이제부터 새 색상들을 추가할 때 모양 계층구조를 변경할 필요가 없으며 그 반대의 경우도 마찬가지입니다.

추상화와 구현

GoF의 디자인 패턴¹은 브리지 패턴 정의의 일부로 추상화 및 구현이라는 용어들을 소개합니다. 저는 위 용어들이 너무 학문적이라

-
1. 'Gang of Four'(사인조)는 디자인 패턴에 관한 1994년 원조 책의 4명의 저자에게 주어진 별명입니다:『GoF의 디자인 패턴: 재사용성을 지닌 객체지향 소프트웨어의 핵심 요소』

그로 인해 패턴이 실제보다 더 복잡하게 들린다고 생각합니다. 그러면 GoF의 책의 난해한 용어들 뒤에 숨겨진 의미를 모양과 색상이 있는 간단한 예를 통해 해독해 보겠습니다.

추상화(인터페이스라고도 함)는 일부 개체(entity)에 대한 상위 수준의 제어 레이어입니다. 이 레이어는 자체적으로 실제 작업을 수행해서는 안 되며, 작업들을 구현 레이어(플랫폼이라고도 함)에 위임해야 합니다.

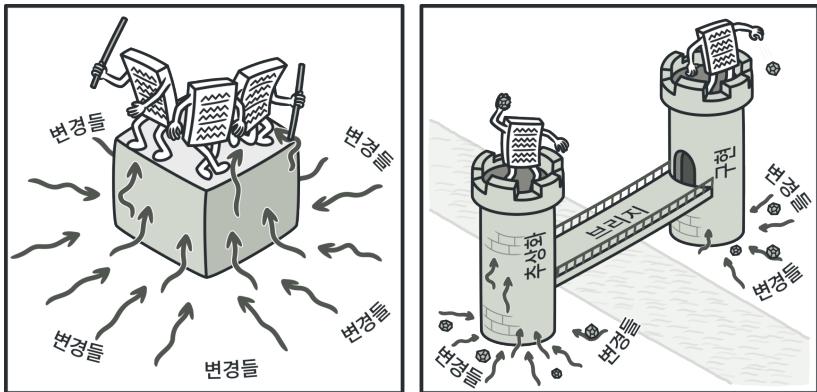
참고로 지금 우리가 이야기하는 것은 당신이 선호하는 프로그래밍 언어의 인터페이스들이나 추상 클래스들이 아닙니다. 이것들은 같은 것이 아닙니다.

실제 앱을 예로 들면 추상화는 그래픽 사용자 인터페이스이며 구현은 그래픽 사용자 인터페이스 레이어가 사용자와 상호작용하여 그 결과로 호출하는 배경 운영 체제 코드(API)입니다.

일반적으로 이러한 앱은 두 가지 독립적인 방향으로 확장할 수 있습니다.

- 다른 여러 가지의 그래픽 사용자 인터페이스를 가진다 (예: 일반 고객 또는 관리자용으로 맞춘 인터페이스들).
- 여러 다른 API들을 지원한다 (예: 맥, 리눅스 및 윈도우에서 앱을 실행할 수 있는 API들).

최악의 경우 이 앱은 수백 개의 조건문들이 코드 전체에 다양한 API와 다양한 유형의 그래픽 사용자 인터페이스들을 연결한 거대한 스파게티 코드 그릇처럼 형성됩니다.

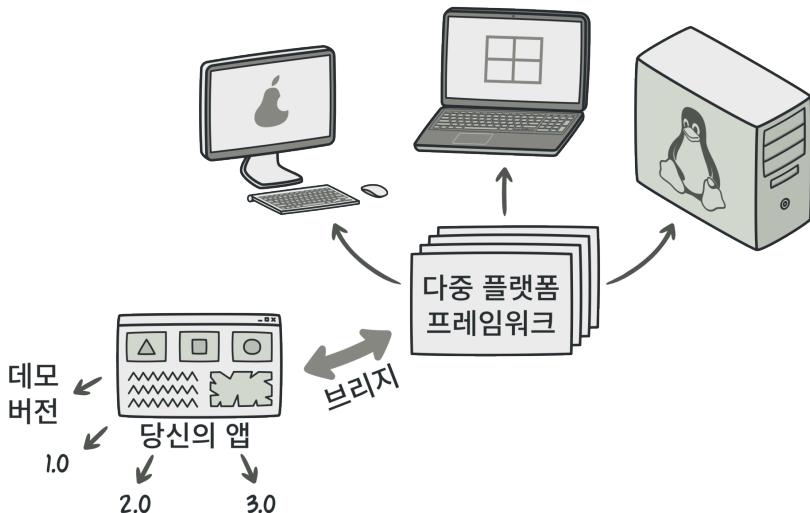


전체를 잘 이해해야 하므로 모놀리식 코드베이스는 간단한 변경을 하는 것조차 매우 어렵습니다. 반면에 잘 정의된 작은 모듈들을 변경하는 것이 훨씬 쉽습니다.

당신은 특정 인터페이스-플랫폼 조합들과 관련된 코드를 별도의 클래스들로 추출하여 이 복잡함에 질서를 부여할 수 있으나, 곧 이러한 클래스들이 많이 있다는 것을 알게 될 것입니다. 새로운 그래픽 사용자 인터페이스를 추가하거나 다른 API를 지원하려면 점점 더 많은 클래스를 생성해야 하므로 클래스 계층구조는 기하급수적으로 성장할 것입니다.

브리지 패턴으로 이 문제를 해결해 봅시다. 브리지 패턴은 클래스들을 두 개의 계층구조로 분리하라고 제안합니다:

- 추상화: 앱의 그래픽 사용자 인터페이스 레이어.
- 구현: 운영 체제의 API.

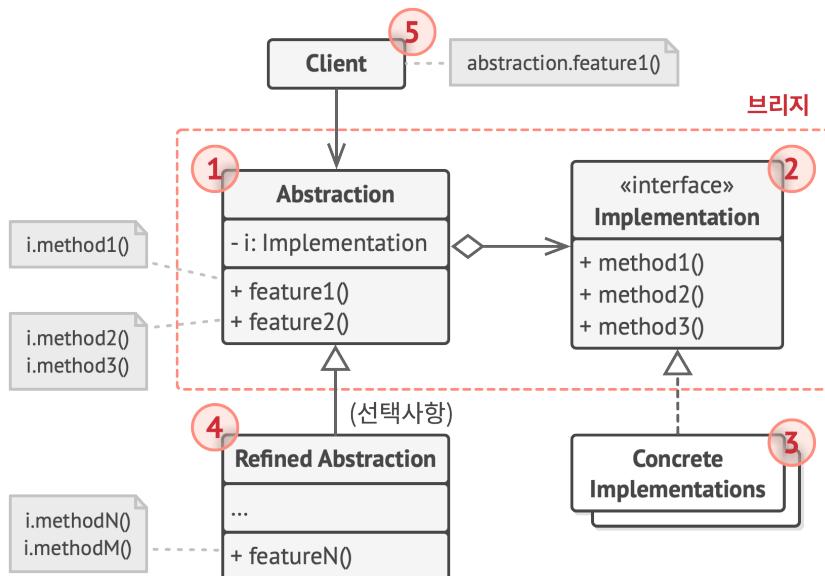


크로스 플랫폼 앱을 구성하는 방법 중 하나.

추상화 객체는 앱의 드러나는 모습을 제어하고 연결된 구현 객체에 실제 작업들을 위임합니다. 서로 다른 구현들은 공통 인터페이스를 따르는 한 상호 호환이 가능하며, 이에 따라 같은 그래픽 사용자 인터페이스는 리눅스와 윈도우에 동시에 작동할 수 있습니다.

따라서 당신은 API 관련 클래스들을 건드리지 않고 그래픽 사용자 인터페이스 클래스들을 변경할 수 있습니다. 그리고 다른 운영 체제에 대한 지원을 추가하려면 구현 계층구조 내에 자식 클래스를 생성하기만 하면 됩니다.

구조



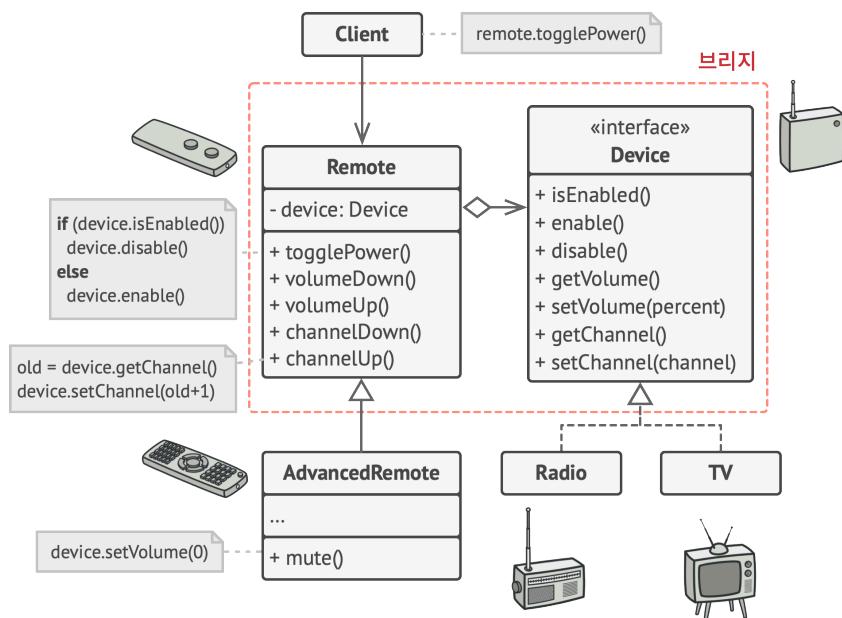
1. **추상화**는 상위 수준의 제어 논리를 제공하며, 구현 객체에 의존해 실제 하위 수준 작업들을 수행합니다.
2. **구현**은 모든 구상 구현들에 공통적인 인터페이스를 선언하며, 추상화는 여기에 선언된 메서드들을 통해서만 구현 객체와 소통할 수 있습니다.

추상화는 구현과 같은 메서드들을 나열할 수 있지만 보통은 구현이 선언한 다양한 원시 작업들에 의존하는 몇 가지 복잡한 행동들을 선언합니다.

3. 구상 구현들에는 플랫폼별 맞춤형 코드가 포함됩니다.

4. 정제된 추상화들은 제어 논리의 변형들을 제공합니다. 그들은 그들의 부모처럼 일반 구현 인터페이스를 통해 다른 구현들과 작업합니다.
5. 일반적으로 클라이언트는 추상화와 작업하는데만 관심이 있습니다. 그러나 추상화 객체를 구현 객체들 중 하나와 연결하는 것도 클라이언트의 역할입니다.

의사코드



원조 클래스의 계층구조는 두 부분으로 나뉩니다: 장치들과 리모컨들.

이 예시는 **브리지** 패턴이 기기와 리모컨을 관리하는 앱의 모듈리식 코드를 나누는 데 어떻게 도움이 되는지 보여줍니다.

Device 클래스들은 구현의 역할을 하는 반면,
Remote 클래스들은 추상화 역할을 합니다.

기초 리모컨 클래스는 이 클래스를 장치 객체와 연결하는 참조 필드를 선언합니다. 모든 리모컨은 일반 장치 인터페이스를 통해 장치들과 작동하므로 같은 리모컨이 여러 장치 유형을 지원할 수 있습니다.

장치 클래스들과 독립적으로 리모컨 클래스들을 개발할 수 있으며, 필요한 것은 새로운 리모컨 자식 클래스를 만드는 것뿐입니다. 예를 들어 기초 리모컨에는 버튼이 두 개뿐일 수 있지만, 추가 터치스크린과 추가 배터리 같은 기능들도 가지도록 확장할 수 있습니다.

클라이언트 코드는 Remote의 생성자를 통해 원하는 유형의 리모컨을 특정 장치 객체와 연결합니다.

```

1 // '추상화'는 두 클래스 계층구조의 '제어' 부분에 대한 인터페이스를 정의하며,
2 // 이것은 '구현' 계층구조의 객체에 대한 참조를 유지하고 모든 실제 작업을 이
3 // 객체에 위임합니다.
4 class RemoteControl is
5   protected field device: Device
6   constructor RemoteControl(device: Device) is
7     this.device = device
8   method togglePower() is
9     if (device.isEnabled()) then
10       device.disable()

```

```

11     else
12         device.enable()
13     method volumeDown() is
14         device.setVolume(device.getVolume() - 10)
15     method volumeUp() is
16         device.setVolume(device.getVolume() + 10)
17     method channelDown() is
18         device.setChannel(device.getChannel() - 1)
19     method channelUp() is
20         device.setChannel(device.getChannel() + 1)
21
22
23 // 이제 추상화 계층구조로부터 클래스들을 장치 클래스들과 독립적으로 확장할 수
24 // 있습니다.
25 class AdvancedRemoteControl extends RemoteControl is
26     method mute() is
27         device.setVolume(0)
28
29
30 // '구현' 인터페이스는 모든 구상 구현 클래스들에 공통적인 메서드를 선언하며, 이는
31 // 추상화의 인터페이스와 일치할 필요가 없습니다. 실제로 두 인터페이스는 완전히 다를
32 // 수 있습니다. 일반적으로 구현 인터페이스는 원시(primitive) 작업들만 제공하는
33 // 반면 추상화는 이러한 원시 작업들을 기반으로 더 상위 수준의 작업들을 정의합니다.
34 interface Device is
35     method isEnabled()
36     method enable()
37     method disable()
38     method getVolume()
39     method setVolume(percent)
40     method getChannel()
41     method setChannel(channel)
42

```

```

43
44 // 모든 장치는 같은 인터페이스를 따릅니다.
45 class Tv implements Device is
46   // ...
47
48 class Radio implements Device is
49   // ...
50
51
52 // 클라이언트 코드 어딘가에...
53 tv = new Tv()
54 remote = new RemoteControl(tv)
55 remote.togglePower()
56
57 radio = new Radio()
58 remote = new AdvancedRemoteControl(radio)

```

💡 적용

💡 브리지 패턴은 당신이 어떤 기능의 여러 변형을 가진 모듈리식 클래스를 나누고 정돈하려 할 때 사용하세요. (예: 클래스가 다양한 데이터베이스 서버들과 작동할 수 있는 경우).

⚡️ 클래스가 성장할수록 그 작동 방식을 파악하기가 더 어려워지고 해당 클래스를 변경하는 데 더더욱 오랜 시간이 걸립니다. 클래스 기능의 여러 변형 중 하나를 변경하려면 클래스 전체에 걸쳐 여러 가지 변경을 수행해야 할 수 있으며, 이를 수행 중 개발자들은 종종 실수하거나 일부 중요한 부작용들을 해결하지 않기도 합니다.

브리지 패턴을 사용하면 모놀리식 클래스를 여러 클래스 계층구조로 나눌 수 있습니다. 그런 다음 각 계층구조의 클래스들을 다른 계층구조들에 있는 클래스들과는 독립적으로 변경할 수 있습니다. 이 접근 방식은 코드의 유지관리를 단순화하고 기존 코드가 손상될 위험을 최소화합니다.

☞ 이 패턴은 여러 직교(독립) 차원에서 클래스를 확장해야 할 때 사용하세요.

↳ 브리지 패턴은 각 차원에 대해 별도의 클래스 계층구조를 추출할 것을 제안합니다. 원래 클래스는 모든 작업을 자체적으로 수행하는 대신 추출된 계층구조들에 속한 객체들에 관련 작업들을 위임합니다.

☞ 브리지 패턴은 런타임(실행시간)에 구현을 전환할 수 있어야 할 때에 사용하세요.

↳ 선택 사항이지만 브리지 패턴을 사용하면 추상화 내부의 구현 객체를 바꿀 수 있으며, 그렇게 하려면 필드에 새 값을 할당하기만 하면 됩니다.

위 항목은 많은 사람이 브리지와 전략 패턴을 혼동하는 주된 이유입니다. 패턴은 클래스의 구조를 설계하는 특정 방법 그 이상의 것이라는 것을 기억하세요. 패턴은 제기되고 있는 문제 및 의도에 대하여도 소통할 수 있습니다.

▣ 구현방법

1. 클래스에서 직교 차원들을 식별하세요. 이러한 독립적인 개념들은 추상화/플랫폼, 도메인/인프라, 프런트엔드/백엔드 또는 인터페이스/구현 등일 수 있습니다.
2. 클라이언트가 필요로 하는 작업들을 확인한 후 기초 추상 클래스에서 정의하세요.
3. 모든 플랫폼들에 제공되어야 하는 작업들을 결정하세요. 그 후 추상화에 필요한 작업들을 일반 구현 인터페이스에서 선언하세요.
4. 도메인의 모든 플랫폼에 대해 구상 구현 클래스들을 생성하되 이 클래스들 모두가 구현 인터페이스를 따르도록 하세요.
5. 추상화 클래스 내에서 구현 유형에 대한 참조 필드를 추가하세요. 추상화는 대부분 작업들을 위 필드에서 참조되는 구현 객체에 위임합니다.
6. 상위 수준 논리의 변형들이 여러 개 있는 경우 기초 추상화 클래스를 확장하여 각 변형에 대해 정제된 추상화들을 만드세요.
7. 클라이언트 코드는 구현 객체를 추상화의 생성자에 전달하여 이 객체를 그 생성자에 연관시켜야 합니다. 그 후에 클라이언트는 구현을 잊어버린 후 추상화 객체와만 작업할 수 있습니다.

⚠️ 장단점

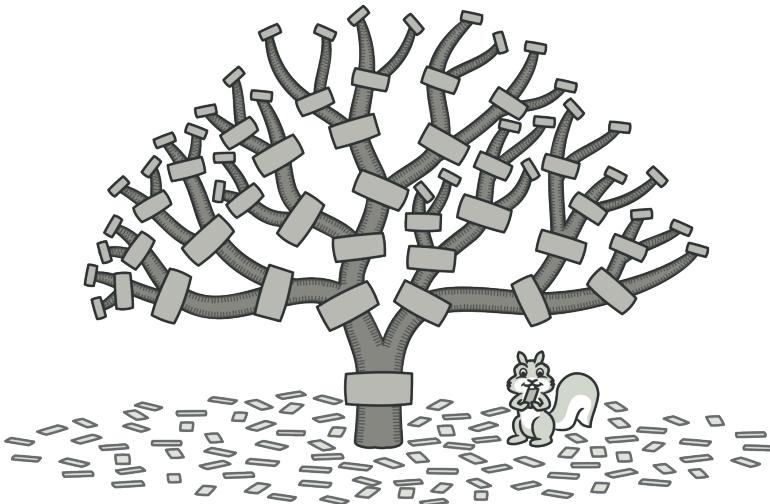
- ✓ 플랫폼 독립적인 클래스들과 앱들을 만들 수 있습니다.
- ✓ 클라이언트 코드는 상위 수준의 추상화를 통해 작동하며, 플랫폼 세부 정보에 노출되지 않습니다.
- ✓ ~~개방/폐쇄 원칙~~. 새로운 추상화들과 구현들을 상호 독립적으로 도입할 수 있습니다.
- ✓ ~~단일 책임 원칙~~. 추상화의 상위 수준 논리와 구현의 플랫폼 세부 정보에 집중할 수 있습니다.
- ✗ 결합도가 높은 클래스에 패턴을 적용하여 코드를 더 복잡하게 만들 수 있습니다.

↔ 다른 패턴과의 관계

- 브리지는 일반적으로 사전에 설계되며, 앱의 다양한 부분을 독립적으로 개발할 수 있도록 합니다. 반면에 어댑터는 일반적으로 기존 앱과 사용되어 원래 호환되지 않던 일부 클래스들이 서로 잘 작동하도록 합니다.
- 브리지, 상태, 전략 패턴은 매우 유사한 구조로 되어 있으며, 어댑터 패턴도 이들과 어느 정도 유사한 구조로 되어 있습니다. 위 모든 패턴은 다른 객체에 작업을 위임하는 합성을 기반으로 합니다. 하지만 이 패턴들은 모두 다른 문제들을 해결합니다. 패턴은 특정 방식으로 코드의 구조를 짜는 레시피에 불과하지

않습니다. 왜냐하면 패턴은 해결하는 문제를 다른 개발자들에게 전달할 수도 있기 때문입니다.

- 당신은 추상 팩토리를 브리지와 함께 사용할 수 있습니다. 이 조합은 브리지에 의해 정의된 어떤 추상화들이 특정 구현들과만 작동할 수 있을 때 유용합니다. 이런 경우에 추상 팩토리는 이러한 관계들을 캡슐화하고 클라이언트 코드에서부터 복잡성을 숨길 수 있습니다.
- 빌더를 브리지와 조합할 수 있습니다. 디렉터 클래스는 추상화의 역할을 하고 다양한 빌더들은 구현의 역할을 합니다.



복합체 패턴

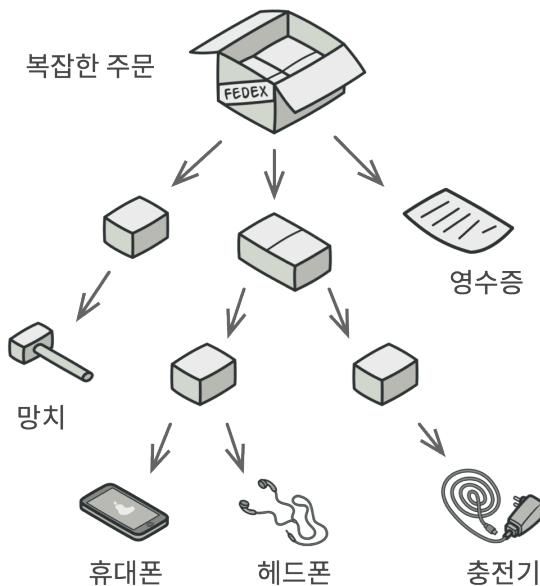
다음 이름으로도 불립니다: 객체 트리, Composite

복합체 패턴은 객체들을 트리 구조들로 구성한 후, 이러한 구조들과 개별 객체들처럼 작업할 수 있도록 하는 구조 패턴입니다.

(?) 문제

복합체 패턴은 앱의 핵심 모델이 트리로 표현될 수 있을 때만 사용하세요.

예를 들어 **제품들**과 **상자들**이라는 두 가지 유형의 객체들이 있다고 가정해 봅시다. **상자**에는 여러 개의 **제품들**과 여러 개의 작은 **상자들**이 포함될 수 있습니다. 이 작은 **상자들**은 또한 일부 **제품들** 또는 더 작은 **상자들** 등을 담을 수 있습니다.



하나의 주문은 여러 제품으로 구성될 수 있는데, 이 제품들은 상자에 포장될 수 있으며, 다시 그 상자들이 더 큰 상자에 포장되는 식으로 이어질 수 있습니다. 그러면 그 전체 구조는 거꾸로 된 나무와 같은 모양으로 형성될 것입니다.

이러한 클래스들을 사용하는 주문 시스템을 만들기로 했다고 가정해 보겠습니다. 주문들에는 포장이 없는 단순한 제품들과 제품들로 채워진 상자들 및 다른 상자들이 포함될 수 있습니다. 그러면 그러한 주문의 총가격을 어떻게 계산하시겠습니까?

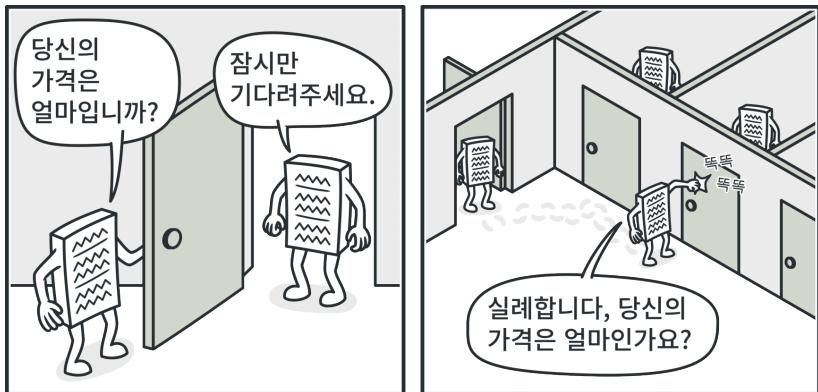
당신은 이 문제에 직접적으로 접근해 볼 수 있습니다: 모든 상자를 푼 후 내부의 모든 제품을 살펴본 다음 가격의 합계를 계산하는 것입니다. 현실 세계에서는 이러한 접근 방법은 가능하나, 프로그램에서 이 작업은 덧셈 루프를 실행하는 것만큼 간단하지 않은데, 그 이유는 덧셈 루프를 실행하기 위해 진행 중인 제품들 및 상자들 의 클래스들, 상자의 중첩 수준 및 기타 복잡한 세부 사항들을 미리 알고 있어야 하기 때문입니다. 이 모든 것이 상자 및 내부 상자들을 열어 포함된 모든 제품 가격의 합계를 계산하는 직접적인 접근 방식을 어렵게 만듭니다.

☺ 해결책

복합체 패턴은 총가격을 계산하는 메서드를 선언하는 공통 인터페이스를 통해 제품들 및 상자들 클래스들과 작업할 것을 제안합니다.

그러면 이 메서드는 어떻게 작동할까요? 제품의 경우 이 메서드는 단순히 제품 가격을 반환합니다. 상자의 경우, 이 메서드는 상자에 포함된 각 항목을 살펴보고 가격을 확인한 뒤 해당 상자의 총 가격을 반환합니다. 만약 이 항목들 중 하나가 더 작은 상자라면, 메서드는 해당 상자의 모든 내부 구성 요소의 가격이 계산될

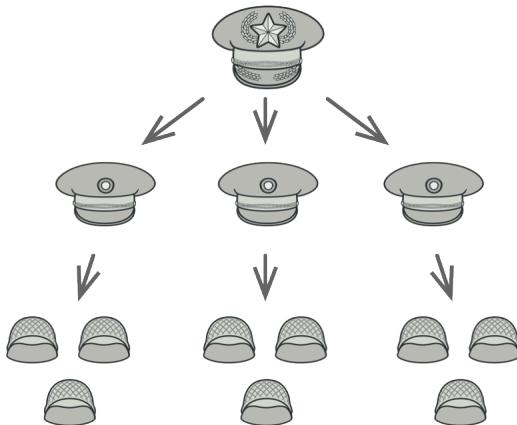
때까지 내용물 등을 살펴봅니다. 메서드는 상자를 다룰 때 최종 가격에 포장 비용 같은 약간의 추가 비용도 추가할 수도 있습니다.



복합체 패턴은 객체 트리의 모든 컴포넌트들에 대해 재귀적으로 행동을 실행할 수 있도록 합니다.

이 접근 방식의 가장 큰 이점은 더 이상 트리를 구성하는 객체들의 구상 클래스들에 대해 신경 쓸 필요도, 또 물건이 단순한 제품인지 내용물이 있는 상자인지 알 필요도 없다는 점입니다. 단순히 공통 인터페이스를 통해 모두 같은 방식으로 처리하시면 됩니다. 당신이 메서드를 호출하면 객체들 자체가 요청을 트리 아래로 전달합니다.

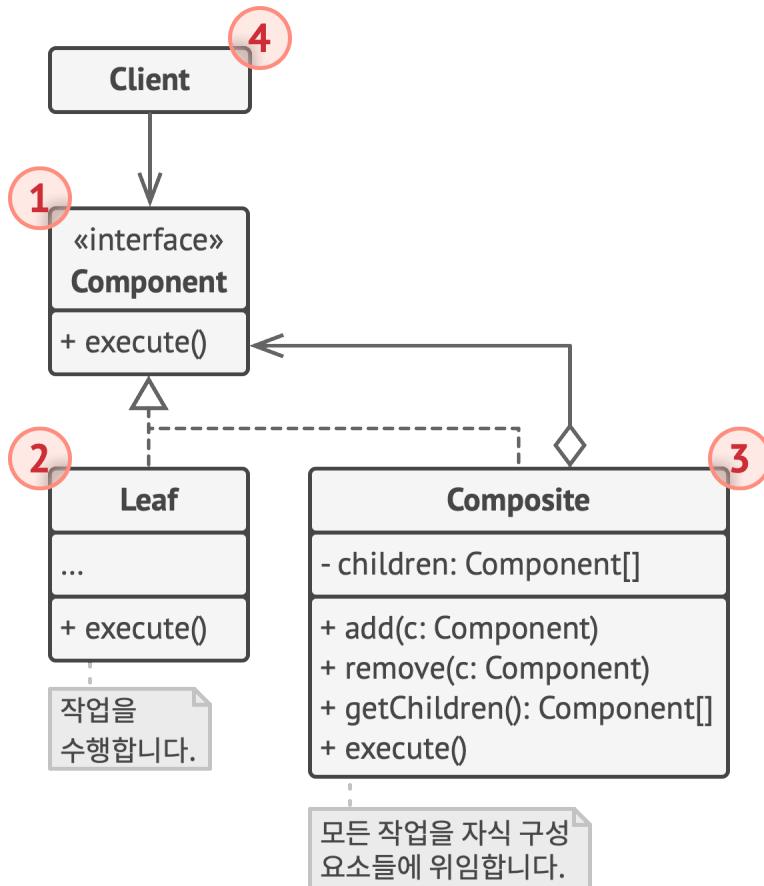
실제상황 적용



군대 구조의 예시.

대부분의 국가에서 군대는 계층구조로 구성되어 있습니다. 군대는 여러 사단으로 구성되며, 사단은 여단의 집합이고, 여단은 소대의 집합이며, 소대는 또 분대로 나누어질 수 있습니다. 마지막으로 분대는 실제 군인들의 작은 집합입니다. 명령들은 계층구조의 최상위에서 내려와 모든 병사가 자신이 수행해야 할 작업을 알게 될 때까지 계층구조의 각 하위 계층으로 전달됩니다.

구조



1. **컴포넌트** 인터페이스는 트리의 단순 요소들과 복잡한 요소들 모두에 공통적인 작업을 설명합니다.
2. **잎은** 트리의 기본 요소이며 하위요소가 없습니다.

일반적으로 잎 컴포넌트들은 작업을 위임할 하위요소가 없어서 대부분의 실제 작업들을 수행합니다.

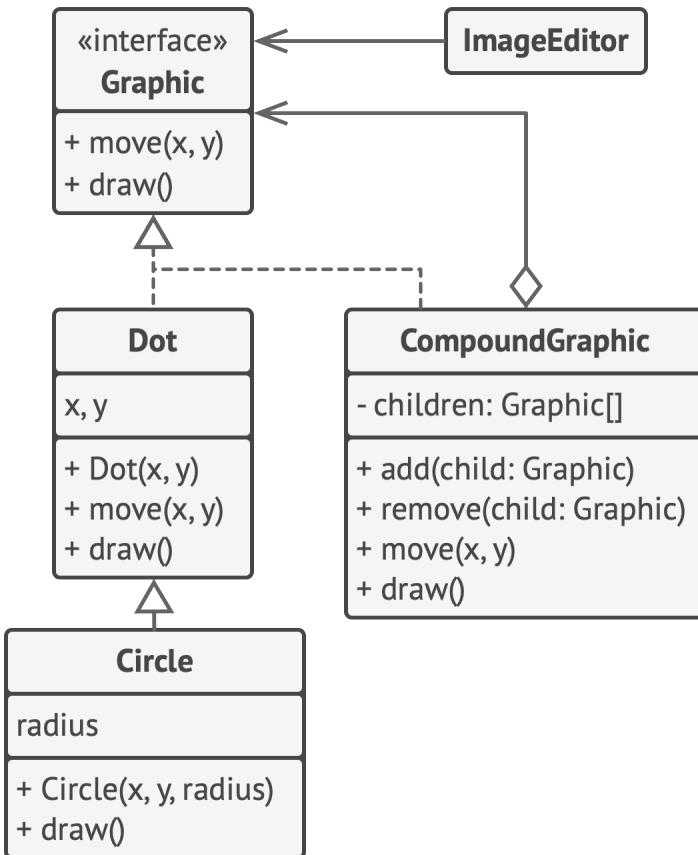
3. **컨테이너**(일명 복합체)는 하위 요소들(잎 또는 기타 컨테이너)이 있는 요소입니다. 컨테이너는 자녀들의 구상 클래스들을 알지 못하며, 컴포넌트 인터페이스를 통해서만 모든 하위 요소들과 함께 작동합니다.

요청을 전달받으면 컨테이너는 작업을 하위 요소들에 위임하고 중간 결과들을 처리한 다음 최종 결과들을 클라이언트에 반환합니다.

4. **클라이언트**는 컴포넌트 인터페이스를 통해 모든 요소들과 작동합니다. 그 결과 클라이언트는 트리의 단순 요소들 또는 복잡한 요소들 모두에 대해 같은 방식으로 작업할 수 있습니다.

의사코드

이 예시에서는 **복합체** 패턴이 어떻게 그래픽 편집기에서 기하학적 모양 쌓기를 구현할 수 있도록 하는지를 살펴보겠습니다.



기하학적 모양 편집기 예시.

`CompoundGraphic` 클래스는 다른 복합 모양들을 포함한 모든 하위 모양으로 구성될 수 있는 컨테이너입니다. 복합 모양은 단순 모양과 같은 메서드들을 보유하나, 자체적으로 무언가를 수행하는 대신 복합 모양은 모든 자녀에게 재귀적으로 요청을 전달하고 이의 결과를 '요약'합니다.

클라이언트 코드는 모든 모양 클래스들에 공통인 단일 인터페이스를 통해 모든 모양과 함께 작동합니다. 따라서 클라이언트는 자신이 단순 모양과 작업하는지 복합 모양과 작업하는지를 알지 못합니다. 또 클라이언트는 매우 복잡한 객체 구조들을 형성하는 구상 클래스들에 결합하지 않고도 해당 구조들과 작업할 수 있습니다.

```

1 // 컴포넌트 인터페이스는 합성 관계의 단순 객체와 복잡한 객체 모두를 위한 공통
2 // 작업들을 선언합니다.
3 interface Graphic is
4     method move(x, y)
5     method draw()
6
7     // 이 클래스는 합성 관계의 최종 객체들을 나타냅니다. 이 객체는 하위 객체들을 가질
8     // 수 없습니다. 일반적으로 실제 작업을 수행하는 것은 이 객체들이며, 복합체 객체들은
9     // 오로지 자신의 하위 컴포넌트에만 작업을 위임합니다.
10 class Dot implements Graphic is
11     field x, y
12
13     constructor Dot(x, y) { ... }
14
15     method move(x, y) is
16         this.x += x, this.y += y
17
18     method draw() is
19         // X와 Y에 점을 그립니다.
20
21     // 모든 컴포넌트 클래스들은 다른 컴포넌트들을 확장할 수 있습니다.
22 class Circle extends Dot is
23     field radius

```

```
24
25     constructor Circle(x, y, radius) { ... }
26
27     method draw() is
28         // X와 Y에 반지름이 R인 원을 그립니다.
29
30     // 복합체 클래스는 자식이 있을 수 있는 복잡한 컴포넌트들을 나타냅니다. 복합체
31     // 객체들은 일반적으로 실제 작업을 자식들에 위임한 다음 결과를 '합산'합니다.
32     class CompoundGraphic implements Graphic is
33         field children: array of Graphic
34
35         // 복합체 객체는 자식 리스트에 단순한 또는 복잡한 다른 컴포넌트들을 추가하거나
36         // 제거할 수 있습니다.
37         method add(child: Graphic) is
38             // 하나의 자식을 자식들의 배열에 추가합니다.
39
40         method remove(child: Graphic) is
41             // 하나의 자식을 자식들의 배열에서 제거합니다.
42
43         method move(x, y) is
44             foreach (child in children) do
45                 child.move(x, y)
46
47         // 복합체는 특정 방식으로 기본 논리를 실행합니다. 복합체는 모든 자식을
48         // 재귀적으로 순회하여 결과들을 수집하고 요약합니다. 복합체의 자식들이 이러한
49         // 호출들을 자신의 자식들 등으로 전달하기 때문에 결과적으로 전체 객체 트리를
50         // 순회하게 됩니다.
51         method draw() is
52             // 1. 각 자식 컴포넌트에 대해:
53             //     - 컴포넌트를 그리세요.
54             //     - 경계 사각형을 업데이트하세요.
55             // 2. 경계 좌표를 사용하여 점선 직사각형을 그리세요.
```

```

56
57
58 // 클라이언트 코드는 기초 인터페이스를 통해 모든 컴포넌트와 함께 작동합니다. 그래야
59 // 클라이언트 코드가 단순한 잎 컴포넌트들과 복잡한 복합체들을 지원할 수 있습니다.
60 class ImageEditor is
61     field all: CompoundGraphic
62
63 method load() is
64     all = new CompoundGraphic()
65     all.add(new Dot(1, 2))
66     all.add(new Circle(5, 3, 10))
67     // ...
68
69 // 선택한 컴포넌트들을 하나의 복잡한 복합체 컴포넌트로 합성합니다.
70 method groupSelected(components: array of Graphic) is
71     group = new CompoundGraphic()
72     foreach (component in components) do
73         group.add(component)
74         all.remove(component)
75     all.add(group)
76     // 모든 컴포넌트가 그려질 것입니다.
77     all.draw()

```

⌚ 적용

 복합체 패턴은 나무와 같은 객체 구조를 구현해야 할 때 사용하세요.

 복합체 패턴은 공통 인터페이스를 공유하는 두 가지 기본 요소 유형들인 단순 잎들과 복합 컨테이너들을 제공합니다. 컨테이너는

잎들과 다른 컨테이너들로 구성될 수 있으며, 이를 통해 나무와 유사한 중첩된 재귀 객체 구조를 구성할 수 있습니다.

 **이 패턴은 클라이언트 코드가 단순 요소들과 복합 요소들을 모두 균일하게 처리하도록 하고 싶을 때 사용하세요.**

 복합체 패턴에 의해 정의된 모든 요소들은 공통 인터페이스를 공유하며, 이 인터페이스를 사용하면 클라이언트는 작업하는 객체들의 구상 클래스에 대해 걱정할 필요가 없습니다.

구현방법

1. 앱의 핵심 모델이 트리 구조로 표현될 수 있는지 확인하세요. 그 후 앱을 단순 요소와 컨테이너로 분해하세요. 컨테이너는 다른 컨테이너들과 단순 요소들을 모두 포함할 수 있어야 합니다.
2. 컴포넌트 인터페이스를 선언하세요. 이 인터페이스 내에는 복잡하고 단순한 컴포넌트 모두에 적합한 메서드들의 리스트를 포함하세요.
3. 단순 요소들을 나타내는 잎 클래스를 생성하세요. 하나의 프로그램에는 여러 개의 서로 다른 잎 클래스들이 있을 수 있습니다.

4. 복잡한 요소들을 나타내는 컨테이너 클래스를 만든 후, 이 클래스에 하위 요소들에 대한 참조를 저장하기 위한 배열 필드를 제공하세요. 배열은 잎과 컨테이너를 모두 저장할 수 있어야 하므로 컴포넌트 인터페이스 유형으로 선언하셔야 합니다.

컴포넌트 인터페이스의 메서드들을 구현하는 동안 컨테이너는 대부분 작업을 하위 요소들에 위임해야 한다는 점을 기억하세요.

5. 마지막으로 컨테이너에서 자식 요소들을 추가 및 제거하는 메서드들을 정의하세요.

이러한 작업은 컴포넌트 인터페이스에서 선언할 수 있으나, 그리하면 잎 클래스의 메서드들이 비어 있을 것이므로 인터페이스 분리 원칙을 위반할 것입니다. 그러나 클라이언트는 트리를 구성할 때도 포함해서 모든 요소를 동등하게 처리할 수 있을 것입니다.

⚠️ 장단점

- ✓ 다형성과 재귀를 당신에 유리하게 사용해 복잡한 트리 구조들과 더 편리하게 작업할 수 있습니다.
- ✓ **개방/폐쇄 원칙**. 객체 트리와 작동하는 기존 코드를 훼손하지 않고 앱에 새로운 요소 유형들을 도입할 수 있습니다.

- ✖ 기능이 너무 다른 클래스들에는 공통 인터페이스를 제공하기 어려울 수 있으며, 어떤 경우에는 컴포넌트 인터페이스를 과도하게 일반화해야 하여 이해하기 어렵게 만들 수 있습니다.

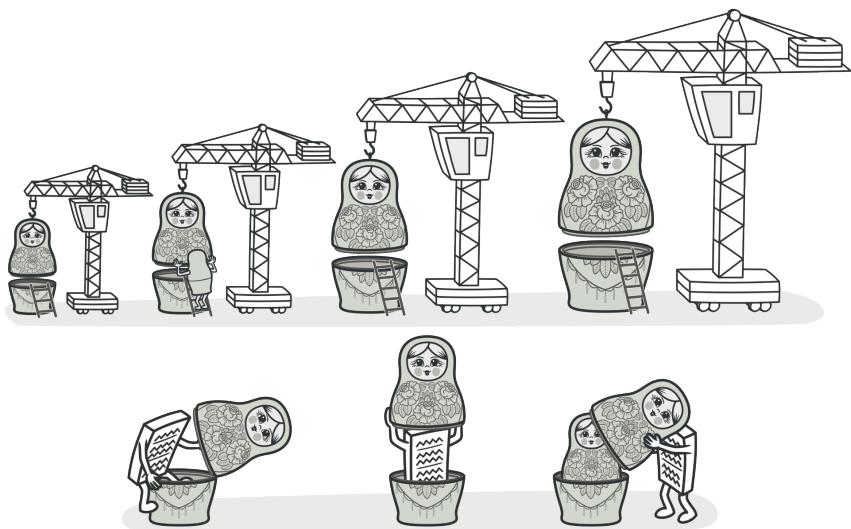
↔ 다른 패턴과의 관계

- 당신은 복잡한 **복합체** 패턴 트리를 생성할 때 **빌더**를 사용할 수 있습니다. 왜냐하면 빌더의 생성 단계들을 재귀적으로 작동하도록 프로그래밍할 수 있기 때문입니다.
- **책임 연쇄** 패턴은 종종 **복합체** 패턴과 함께 사용됩니다. 그러면 잎 컴포넌트가 요청을 받으면 해당 요청을 모든 부모 컴포넌트들의 체인을 통해 객체 트리의 뿌리(root)까지 전달할 수 있습니다.
- 당신은 **반복자들을** 사용하여 **복합체** 패턴 트리들을 순회할 수 있습니다.
- 당신은 **비지터** 패턴을 사용하여 **복합체** 패턴 트리 전체를 대상으로 작업을 수행할 수 있습니다.
- RAM을 절약하기 위하여 **복합체** 패턴 트리의 공유된 잎 노드들을 **플라이웨이트들**로 구현할 수 있습니다.
- **복합체** 패턴 및 **데코레이터**는 둘 다 구조 디어그램이 유사합니다. 왜냐하면 둘 다 재귀적인 합성에 의존하여 하나 또는 불특정 다수의 객체들을 정리하기 때문입니다.

데코레이터는 복합체 패턴과 비슷하지만, 자식 컴포넌트가 하나만 있습니다. 또 다른 중요한 차이점은 데코레이터는 래핑된 객체에 추가 책임들을 추가하는 반면 복합체 패턴은 자신의 자식들의 결과를 '요약'하기만 합니다.

그러나 패턴들은 서로 협력할 수도 있습니다: 데코레이터를 사용하여 복합체 패턴 트리의 특정 객체의 행동을 확장할 수 있습니다.

- **데코레이터** 및 **복합체** 패턴을 많이 사용하는 디자인들은 **프로토타입**을 사용하면 종종 이득을 볼 수 있습니다. 프로토타입 패턴을 적용하면 복잡한 구조들을 처음부터 다시 건축하는 대신 복제할 수 있기 때문입니다.



데코레이터 패턴

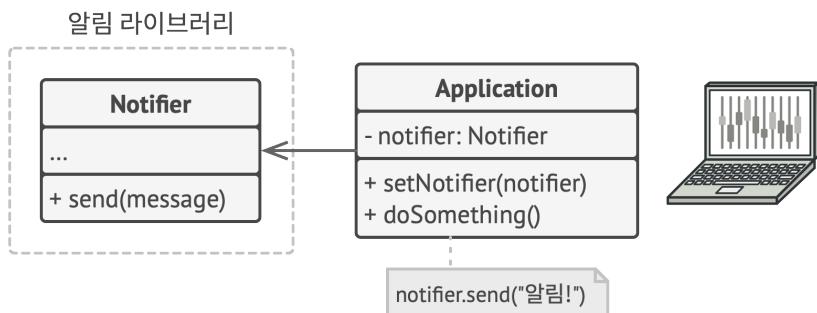
다음 이름으로도 불립니다: 래퍼 (Wrapper), Decorator

데코레이터는 객체들을 새로운 행동들을 포함한 특수 래퍼 객체들 내에 넣어서 위 행동들을 해당 객체들에 연결시키는 구조적 디자인 패턴입니다.

(?): 문제

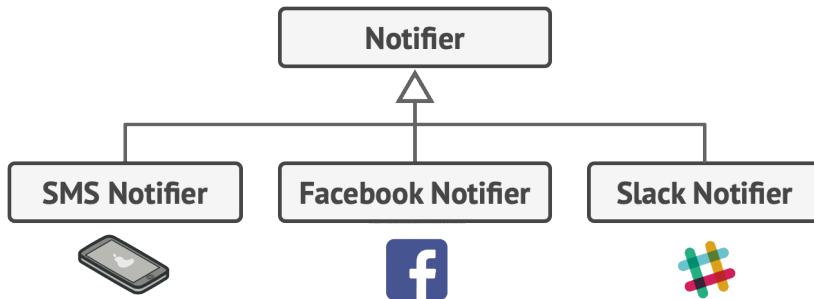
당신이 알림 라이브러리를 만들고 있다고 상상해 보세요. 이 알림 라이브러리의 목적은 다른 프로그램들이 사용자들에게 중요한 이벤트들에 대해 알릴 수 있도록 하는 것입니다.

이 라이브러리의 초기 버전은 `Notifier` (알림자) 클래스를 기반으로 했으며, 이 클래스에는 몇 개의 필드들, 하나의 생성자 그리고 단일 `send` (전송) 메서드만 있었습니다. 이 메서드는 클라이언트로부터 메시지 인수를 받은 후 그 메세지를 알림자의 생성자를 통해 알림자에게 전달된 이메일 목록으로 보낼 수 있습니다. 또 클라이언트 역할을 한 타사 앱은 알림자 객체를 한번 생성하고 설정한 후 중요한 일이 발생할 때마다 사용하게 되어 있었습니다.



프로그램은 알림자 클래스를 사용하여 미리 정의된 이메일들의 집합에 중요한 이벤트에 대한 알림을 보낼 수 있습니다.

당신은 어느 시점에서 라이브러리 사용자들이 이메일 알림 이상을 기대한다는 것을 알게 됩니다. 그들 중 많은 사용자는 중요한 사안에 대한 SMS 문자 메시지를 받고 싶어 했고, 다른 사용자들은 페이스북 알림을 받고 싶어 했으며 기업 사용자들은 슬랙 알림을 받고 싶어 했습니다.



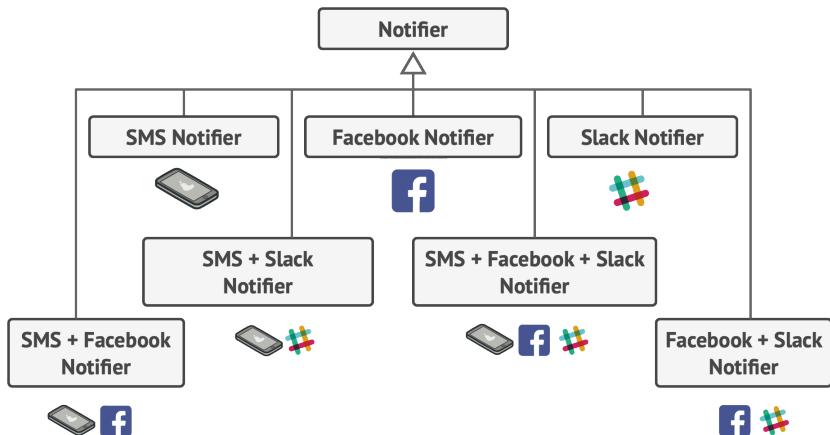
각 알림 유형은 알림자의 자식 클래스로 구현됩니다.

이는 표면상 별로 어렵지 않아 보입니다. 당신은 **Notifier** (알림자) 클래스를 확장하고 추가 알림 메서드들을 새 자식 클래스들에 넣어 이제 클라이언트가 사용자가 원하는 알림 클래스를 인스턴스화하고 모든 추가 알림에 사용하도록 앱을 설계했습니다.

그런데 누군가 당신에게 물었습니다. '여러 유형의 알림을 한 번에 사용할 수는 없나요? 집에 불이라도 난다면 사용자들은 모든 채널에서 정보를 받고 싶어 할 겁니다.'

이 문제를 해결하기 위해 당신은 하나의 클래스 내에서 여러 알림 메서드를 합성한 특수 자식 클래스들을 만들었으나, 이 접근

방식은 라이브러리 코드뿐만 아니라 클라이언트 코드도 엄청나게 부풀릴 것이라는 사실이 금세 명백해졌습니다.



자식 클래스들의 합성으로 인한 클래스 수의 폭발

이제 당신은 알림 클래스들의 수가 지나치게 많아지지 않도록 알림 클래스들을 구성하는 다른 방법을 찾아야 합니다.

😊 해결책

객체의 동작을 변경해야 할 때 가장 먼저 고려되는 방법은 클래스의 확장입니다. 그러나 상속에는 당신이 심각하게 주의해야 할 몇 가지 사항들이 있습니다.

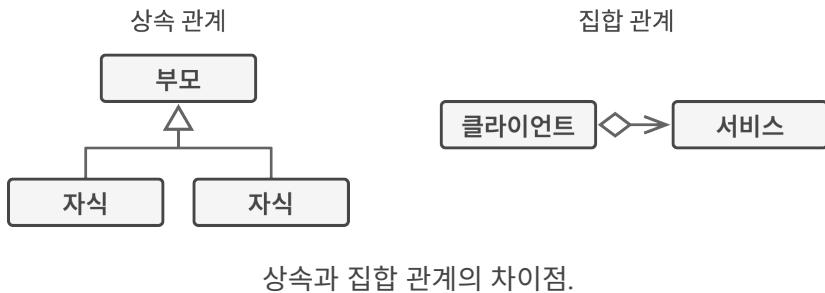
- 상속은 정적입니다: 당신은 런타임(실행시간) 때 기존 객체의 행동을 변경할 수 없습니다. 당신은 전체 객체를 다른 자식 클래스에서 생성된 다른 객체로만 바꿀 수 있습니다.

- 자식 클래스는 하나의 부모 클래스만 가질 수 있습니다. 대부분 언어에서의 상속은 클래스가 동시에 여러 클래스의 행동을 상속하도록 허용하지 않습니다.

이러한 주의 사항을 극복하는 방법의 하나는 상속 대신 집합 관계 또는 합성¹을 사용하는 것입니다. 두 대안 모두 거의 같은 방식으로 작동합니다: 집합 관계에서는 한 객체가 다른 객체에 대한 참조를 갖고 일부 작업을 위임하는 반면, 상속을 사용하면 객체 자체가 부모 클래스에서 행동을 상속한 후 해당 작업을 수행할 수 있습니다.

이 새로운 접근 방식을 사용하면 연결된 '도우미' 객체를 다른 객체로 쉽게 대체하여 런타임 때 컨테이너의 행동을 변경할 수 있습니다. 객체는 여러 클래스의 행동들을 사용할 수 있고, 여러 객체에 대한 참조들이 있으며 이 객체들에 모든 종류의 작업을 위임합니다. 집합 관계/합성은 데코레이터를 포함한 많은 디자인 패턴의 핵심 원칙입니다. 그러면 이제 다시 이 패턴에 대하여 살펴봅시다.

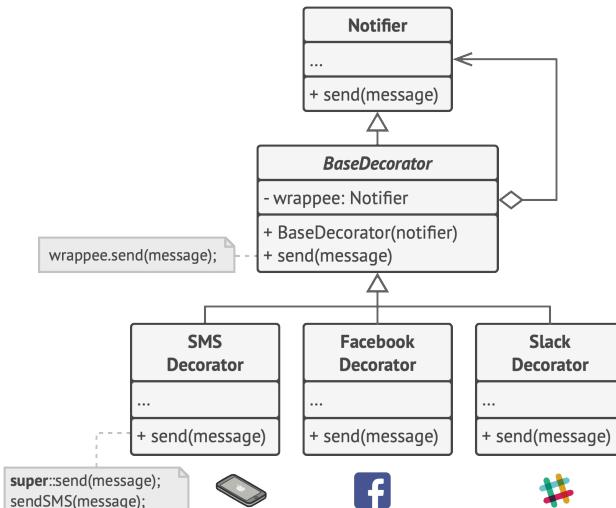
-
1. **집합 관계:** 객체 A는 객체 B를 포함합니다; B는 A 없이 생존할 수 있습니다.
합성: 객체 A는 객체 B로 구성됩니다; A는 B의 수명 주기를 관리합니다; B는 A 없이 생존할 수 없습니다.



'래퍼'는 패턴의 주요 아이디어를 명확하게 표현하는 데코레이터 패턴의 별명입니다. 래퍼는 일부 대상 객체와 연결할 수 있는 객체입니다. 래퍼에는 대상 객체와 같은 메서드들의 집합이 포함되어 있으며, 래퍼는 자신이 받는 모든 요청을 대상 객체에 위임합니다. 그러나 래퍼는 이 요청을 대상에 전달하기 전이나 후에 무언가를 수행하여 결과를 변경할 수 있습니다.

그러면 간단한 래퍼는 언제 진정한 데코레이터가 될 수 있을까요? 앞서 언급했듯이 래퍼는 래핑된 객체와 같은 인터페이스를 구현합니다. 그러므로 클라이언트의 관점에서 이러한 객체들은 같습니다. 이제 래퍼의 참조 필드가 해당 인터페이스를 따르는 모든 객체를 받도록 하세요. 이렇게 하면 여러 래퍼로 객체를 포장해서 모든 래퍼들의 합성된 행동들을 객체에 추가할 수 있습니다.

이제 당신의 알림 라이브러리에서 기초 `Notifier` 클래스 내에 있는 간단한 이메일 알림 행동은 그대로 두고 다른 모든 알림 메서드들을 데코레이터로 바꾸어 봅시다.



다양한 알림 메서드들이 데코레이터가 됩니다.

클라이언트 코드는 기초 알림자 객체를 클라이언트의 요구사항들과 일치하는 데코레이터들의 집합으로 래핑해야 합니다. 위 결과 객체들은 스택으로 구성됩니다.

```

stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
  
```

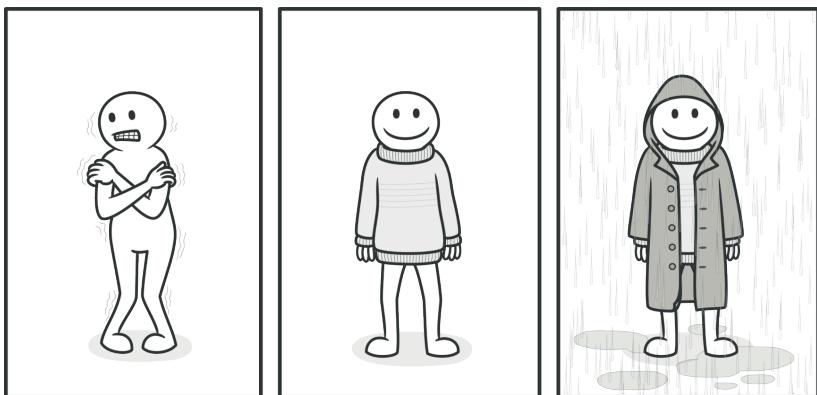


앱들은 알림 데코레이터들의 복잡한 스택들을 설정할 수 있습니다.

스택의 마지막 데코레이터는 실제로 클라이언트와 작업하는 객체입니다. 모든 데코레이터들은 기초 알림자와 같은 인터페이스를 구현하므로 나머지 클라이언트 코드는 자신이 '순수한' 알림자 객체와 작동하든 데코레이터로 장식된 알림자 객체와 함께 작동하든 상관하지 않습니다.

메시지 형식 지정 또는 수신자 리스트 작성과 같은 다른 행동들에도 같은 접근 방식을 적용할 수 있습니다. 클라이언트는 객체가 다른 객체들과 같은 인터페이스를 따르는 한 객체를 모든 사용자 지정 데코레이터로 장식할 수 있습니다.

🚗 실제상황 적용

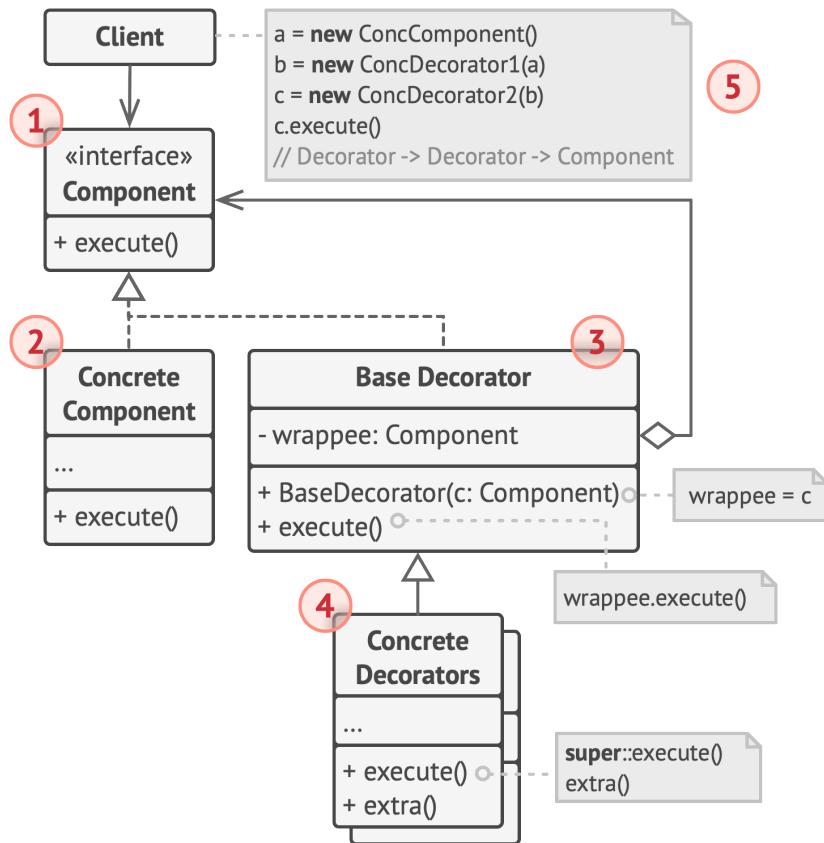


여러 벌의 옷을 입으면 복합 효과를 얻을 수 있습니다.

옷을 입는 것은 데코레이터 패턴을 사용하는 예입니다. 당신은 추울 때 스웨터로 몸을 감쌉니다. 스웨터를 입어도 춥다면 위에 재킷을 입고, 또 비가 오면 비옷을 입습니다. 이 모든 옷은 기초

행동을 '확장'하지만, 당신의 일부가 아니기에 필요하지 않을 때마다 옷을 쉽게 벗을 수 있습니다.

구조

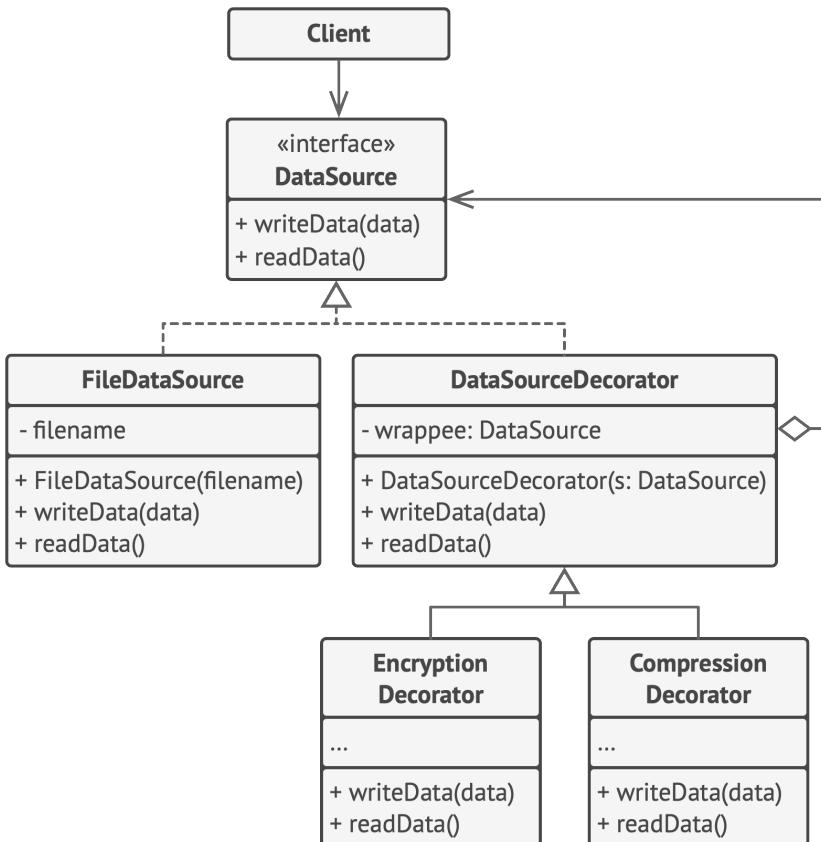


1. **컴포넌트는 래퍼들과 래핑된 객체들 모두에 대한 공통 인터페이스를 선언합니다.**

2. **구상 컴포넌트**는 래핑되는 객체들의 클래스이며, 그는 기본 행동들을 정의하고 해당 기본 행동들은 데코레이터들이 변경할 수 있습니다.
3. **기초 데코레이터** 클래스에는 래핑된 객체를 참조하기 위한 필드가 있습니다. 필드의 유형은 구상 컴포넌트들과 구상 데코레이터들을 모두 포함할 수 있도록 컴포넌트 인터페이스로 선언되어야 합니다. 그 후 기초 데코레이터는 모든 작업들을 래핑된 객체에 위임합니다.
4. **구상 데코레이터들**은 컴포넌트들에 동적으로 추가될 수 있는 추가 행동들을 정의합니다. 그들은 기초 데코레이터의 메서드를 오버라이드(재정의)하고 해당 행동을 부모 메서드를 호출하기 전이나 후에 실행합니다.
5. **클라이언트**는 아래에 언급한 데코레이터들이 컴포넌트 인터페이스를 통해 모든 객체와 작동하는 한 컴포넌트들을 여러 계층의 데코레이터들로 래핑할 수 있습니다.

의사코드

이 예시에서 **데코레이터** 패턴은 민감한 데이터를 해당 데이터를 실제로 사용하는 코드와 별도로 압축하고 암호화할 수 있도록 합니다.



암호화 및 압축화 데코레이터들의 예.

이 애플리케이션은 데이터 소스 객체를 한 쌍의 데코레이터로 래핑합니다. 두 래퍼 모두 디스크에 데이터를 쓰고 읽는 방식들을 변경합니다.

- 데이터가 **디스크에 기록**되기 직전에 데코레이터들은 데이터를 암호화하고 압축합니다. 원래 클래스는 위 변경 사항에 대해 알지 못한 채 암호화되고 보호된 데이터를 파일에 씁니다.

- 데이터는 **디스크에서 읽힌 직후** 같은 데코레이터들을 거쳐 가며, 이 데코레이터들은 데이터의 압축을 풀고 디코딩합니다.

데코레이터와 데이터 소스 클래스는 같은 인터페이스를 구현하므로 클라이언트 코드에서 모두 상호 교환이 가능합니다.

```

1 // 컴포넌트 인터페이스는 데코레이터들이 변경할 수 있는 작업들을 정의합니다.
2 interface DataSource is
3     method writeData(data)
4     method readData():data
5
6 // 구상 컴포넌트들은 작업들에 대한 디폴트 구현들을 제공합니다. 프로그램에는 이러한
7 // 클래스들의 여러 변형이 있을 수 있습니다.
8 class FileDataSource implements DataSource is
9     constructor FileDataSource(filename) { ... }
10
11    method writeData(data) is
12        // 파일에 데이터를 씁니다.
13
14    method readData():data is
15        // 파일에서 데이터를 읽으세요.
16
17 // 기초 데코레이터 클래스는 다른 컴포넌트들과 같은 인터페이스를 따릅니다. 이
18 // 클래스의 주목적은 모든 구상 데코레이터에 대한 래핑 인터페이스를 정의하는
19 // 것입니다. 래핑 코드의 디폴트 구현에는 래핑된 컴포넌트를 저장하기 위한 필드와
20 // 이를 초기화하는 수단들이 포함될 수 있습니다.
21 class DataSourceDecorator implements DataSource is
22     protected field wrappee: DataSource
23
24     constructor DataSourceDecorator(source: DataSource) is

```

```

25     wrappee = source
26
27     // 기초 데코레이터는 단순히 모든 작업을 래핑된 컴포넌트에 위임합니다. 구상
28     // 데코레이터들에는 추가 행동들이 추가될 수 있습니다.
29     method writeData(data) is
30         wrappee.writeData(data)
31
32     // 구상 데코레이터들은 래핑된 객체를 직접 호출하는 대신 부모의 작업 구현을
33     // 호출할 수 있습니다. 이 접근 방식은 데코레이터 클래스들의 확장을
34     // 단순화합니다.
35     method readData():data is
36         return wrappee.readData()
37
38     // 구상 데코레이터는 래핑된 객체에 메서드를 호출해야 하지만 결과에 자기 것을 뭔가를
39     // 추가할 수 있습니다. 데코레이터들은 래핑된 객체에 대한 호출 전 또는 후에 추가된
40     // 행동을 실행할 수 있습니다.
41     class EncryptionDecorator extends DataSourceDecorator is
42         method writeData(data) is
43             // 1. 전달된 데이터를 암호화합니다.
44             // 2. 암호화된 데이터를 wrappee(래핑된)의 writeData(데이터 쓰기)
45             // 메서드에 전달합니다.
46
47         method readData():data is
48             // 1. wrappee(래핑된)의 readData 메서드에서 데이터를 가져옵니다.
49             // 2. 암호화되어 있다면 암호 해독을 시도하세요.
50             // 3. 결과를 반환하세요.
51
52     // 여러 계층의 데코레이터들로 객체들을 래핑할 수 있습니다.
53     class CompressionDecorator extends DataSourceDecorator is
54         method writeData(data) is
55             // 1. 전달된 데이터를 압축하세요.
56             // 2. 압축된 데이터를 wrappee(래핑된)의 writeData 메서드에

```

```
57      // 전달하세요.
```

```
58
```

```
59 method readData():data is
60     // 1. wrappee(래핑된)의 readData(데이터 읽기) 메서드에서 데이터를
61     // 가져오세요.
62     // 2. 압축되어 있으면 압축을 풀어보세요.
63     // 3. 결과를 반환하세요.
```

```
64
```

```
65
```

```
66 // 옵션 1. 데코레이터 조립의 간단한 예시.
```

```
67 class Application is
68     method dumbUsageExample() is
69         source = new FileDataSource("somefile.dat")
70         source.writeData(salaryRecords)
71         // 대상 파일은 일반 데이터로 작성되었습니다.
```

```
72
```

```
73         source = new CompressionDecorator(source)
74         source.writeData(salaryRecords)
75         // 대상 파일은 압축된 데이터로 작성되었습니다.
```

```
76
```

```
77         source = new EncryptionDecorator(source)
78         // 이제 소스 변수에 다음이 포함됩니다. 암호화 > 압축 > 파일 데이터 소스
79         source.writeData(salaryRecords)
80         // 파일은 압축 및 암호화된 데이터로 작성되었습니다.
```

```
81
```

```
82
```

```
83 // 옵션 2. 외부 데이터 소스를 사용하는 클라이언트 코드. SalaryManager 객체들은
84 // 데이터 저장 세부 사항들을 알지도 못하고 신경 쓰지도 않습니다. 이들은 앱 설정
85 // 프로그램에서 받은 사전 구성된 데이터 소스로 작업합니다.
```

```
86 class SalaryManager is
87     field source: DataSource
```

```
88
```

```

99 // 앱은 설정 또는 환경에 따라 런타임 때 다양한 데코레이터 스택들을 조합할 수
100 // 있습니다.
101 class ApplicationConfigurator is
102     method configurationExample() is
103         source = new FileDataSource("salary.dat")
104         if (enabledEncryption)
105             source = new EncryptionDecorator(source)
106         if (enabledCompression)
107             source = new CompressionDecorator(source)
108
109         logger = new SalaryManager(source)
110         salary = logger.load()
111         // ...

```

💡 적용

 데코레이터 패턴은 이 객체들을 사용하는 코드를 훼손하지 않으면서 런타임에 추가 행동들을 객체들에 할당할 수 있어야 할 때 사용하세요.

⚡ 데코레이터는 비즈니스 로직을 계층으로 구성하고, 각 계층에 데코레이터를 생성하고 런타임에 이 로직의 다양한 조합들을 객체들을 구성할 수 있도록 합니다. 이러한 모든 객체가 공통 인터페이스를 따르기 때문에 클라이언트 코드는 해당 모든 객체를 같은 방식으로 다룰 수 있습니다.

⚡ 이 패턴은 상속을 사용하여 객체의 행동을 확장하는 것이 어색하거나 불가능할 때 사용하세요.

⚡ 많은 프로그래밍 언어에는 클래스의 추가 확장을 방지하는 데 사용할 수 있는 `final` 키워드가 있습니다. Final 클래스의 경우 기존 행동들을 재사용할 수 있는 유일한 방법은 데코레이터 패턴을 사용하여 클래스를 자체 래퍼로 래핑하는 것입니다.

▣ 구현방법

1. 당신의 비즈니스 도메인이 여러 선택적 계층으로 감싸진 기본 컴포넌트로 표시될 수 있는지 확인하세요.
2. 기본 컴포넌트와 선택적 계층들 양쪽에 공통적인 메서드들이 무엇인지 파악하세요. 그곳에 컴포넌트 인터페이스를 만들고 해당 메서드들을 선언하세요.
3. 구상 컴포넌트 클래스를 만든 후 그 안에 기초 행동들을 정의하세요.

4. 기초 데코레이터 클래스를 만드세요. 이 클래스에는 래핑된 객체에 대한 참조를 저장하기 위한 필드가 있어야 합니다. 이 필드는 데코레이터들 및 구상 컴포넌트들과의 연결을 허용하기 위하여 컴포넌트 인터페이스 유형으로 선언하셔야 합니다. 기초 데코레이터는 모든 작업을 래핑된 객체에 위임해야 합니다.
5. 모든 클래스들이 컴포넌트 인터페이스를 구현하도록 하세요.
6. 기초 데코레이터를 확장하여 구상 데코레이터들을 생성하세요. 구상 데코레이터는 항상 부모 메서드 호출 전 또는 후에 행동들을 실행해야 합니다. (부모 메서드는 항상 래핑된 객체에 작업을 위임합니다).
7. 데코레이터들을 만들고 이러한 데코레이터들을 클라이언트가 필요로 하는 방식으로 구성하는 일은 반드시 클라이언트 코드가 맡아야 합니다.

⚠️ 장단점

- ✓ 새 자식 클래스를 만들지 않고도 객체의 행동을 확장할 수 있습니다.
- ✓ 런타임에 객체들에서부터 책임들을 추가하거나 제거할 수 있습니다.
- ✓ 객체를 여러 데코레이터로 래핑하여 여러 행동들을 합성할 수 있습니다.

- ✓ 단일 책임 원칙. 다양한 행동들의 여러 변형들을 구현하는 모놀리식 클래스를 여러 개의 작은 클래스들로 나눌 수 있습니다.
- ✗ 래퍼들의 스택에서 특정 래퍼를 제거하기가 어렵습니다.
- ✗ 데코레이터의 행동이 데코레이터 스택 내의 순서에 의존하지 않는 방식으로 데코레이터를 구현하기가 어렵습니다.
- ✗ 계층들의 초기 설정 코드가 보기 흉할 수 있습니다.

↔ 다른 패턴과의 관계

- 어댑터는 기존 객체의 인터페이스를 변경하는 반면 데코레이터는 객체를 해당 객체의 인터페이스를 변경하지 않고 향상합니다. 또한 데코레이터는 어댑터를 사용할 때는 불가능한 재귀적 합성을 지원합니다.
- 어댑터는 다른 인터페이스를, 프록시는 같은 인터페이스를, 데코레이터는 향상된 인터페이스를 래핑된 객체에 제공합니다.
- 책임 연쇄 패턴과 데코레이터는 클래스 구조가 매우 유사합니다. 두 패턴 모두 실행을 일련의 객체들을 통해 전달할 때 재귀적인 합성에 의존하나, 몇 가지 결정적인 차이점이 있습니다.

책임 연쇄 패턴 핸들러들은 서로 독립적으로 임의의 작업을 실행할 수 있으며, 또한 해당 요청을 언제든지 더 이상 전달하지 않을 수 있습니다. 반면에 다양한 데코레이터들은 객체의 행동을

확장하며 동시에 이러한 행동을 기초 인터페이스와 일관되게 유지할 수 있습니다. 또한 데코레이터들은 요청의 흐름을 중단할 수 없습니다.

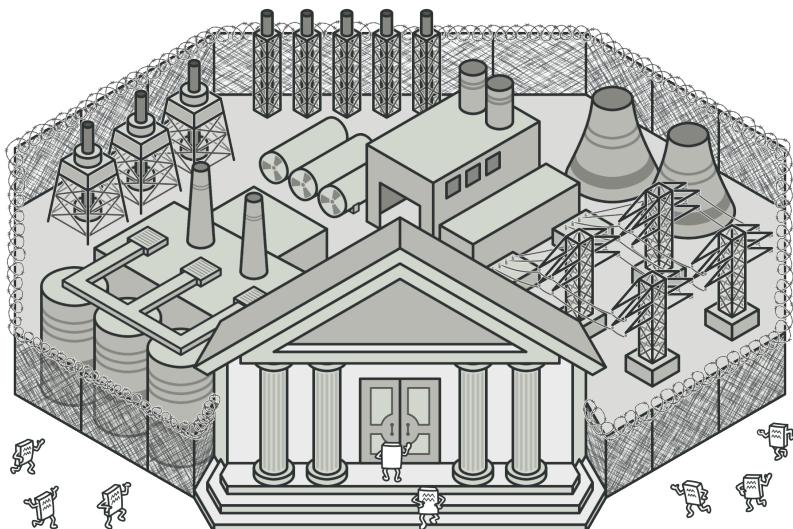
- **복합체** 패턴 및 **데코레이터**는 둘 다 구조 디어그램이 유사합니다. 왜냐하면 둘 다 재귀적인 합성에 의존하여 하나 또는 불특정 다수의 객체들을 정리하기 때문입니다.

데코레이터는 복합체 패턴과 비슷하지만, 자식 컴포넌트가 하나만 있습니다. 또 다른 중요한 차이점은 데코레이터는 래핑된 객체에 추가 책임들을 추가하는 반면 복합체 패턴은 자신의 자식들의 결과를 '요약'하기만 합니다.

그러나 패턴들은 서로 협력할 수도 있습니다: 데코레이터를 사용하여 복합체 패턴 트리의 특정 객체의 행동을 확장할 수 있습니다.

- **데코레이터** 및 **복합체** 패턴을 많이 사용하는 디자인들은 **프로토타입**을 사용하면 종종 이득을 볼 수 있습니다. 프로토타입 패턴을 적용하면 복잡한 구조들을 처음부터 다시 건축하는 대신 복제할 수 있기 때문입니다.
- **데코레이터**는 객체의 피부를 변경할 수 있고 **전략** 패턴은 객체의 내장을 변경할 수 있다고 비유할 수 있습니다.

- 데코레이터와 프록시의 구조는 비슷하나 이들의 의도는 매우 다릅니다. 두 패턴 모두 한 객체가 일부 작업을 다른 객체에 위임해야 하는 합성 원칙을 기반으로 합니다. 이 두 패턴의 차이점은 프록시는 일반적으로 자체적으로 자신의 서비스 객체의 수명 주기를 관리하는 반면 데코레이터의 합성은 항상 클라이언트에 의해 제어된다는 점입니다.



퍼사드 패턴

다음 이름으로도 불립니다: Facade

퍼사드 패턴은 라이브러리에 대한, 프레임워크에 대한 또는 다른 클래스들의 복잡한 집합에 대한 단순화된 인터페이스를 제공하는 구조적 디자인 패턴입니다.

(:(문제

정교한 라이브러리나 프레임워크에 속하는 광범위한 객체들의 집합으로 당신의 코드를 작동하게 만들어야 한다고 상상해 봅시다. 일반적으로, 당신은 이러한 객체들을 모두 초기화하고, 종속성 관계들을 추적하고, 올바른 순서로 메서드들을 실행하는 등의 작업을 수행해야 합니다.

그 결과 당신의 클래스들의 비즈니스 로직이 타사 클래스들의 구현 세부 사항들과 밀접하게 결합하여 코드를 이해하고 유지 관리하기가 어려워집니다.

(:) 해결책

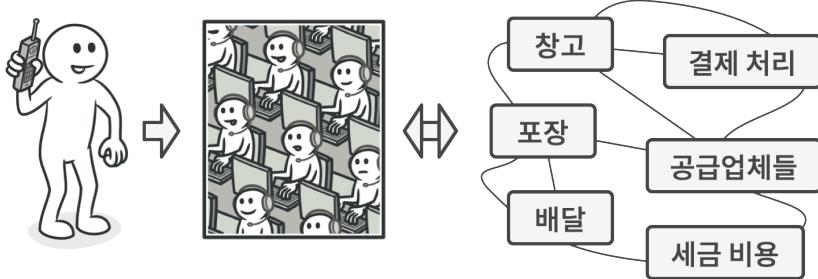
퍼사드는 움직이는 부분이 많이 포함된 복잡한 하위 시스템에 대한 간단한 인터페이스를 제공하는 클래스입니다. 하위 시스템과 직접 작업하는 것과 비교하면 퍼사드는 제한된 기능성을 제공합니다. 하지만 퍼사드에는 클라이언트들이 정말로 중요하게 생각하는 기능들만 포함됩니다.

퍼사드는 당신의 앱을 수십 가지의 기능이 있는 정교한 라이브러리와 통합해야 하지만 그 기능의 극히 일부만을 필요로 할 때 편리합니다.

예를 들어, 고양이가 나오는 짤막하고 재미있는 영상을 소셜 미디어에 올리는 어떤 앱은 전문적인 비디오 변환 라이브러리를

사용할 수 있습니다. 그러나 이 앱이 이 라이브러리로부터 필요로 하는 것은 `encode(filename, format)` 단일 메서드가 있는 클래스뿐입니다. 이러한 클래스를 만든 후 비디오 변환 라이브러리와 연결하면 당신은 당신의 첫 번째 퍼사드를 소유하게 됩니다.

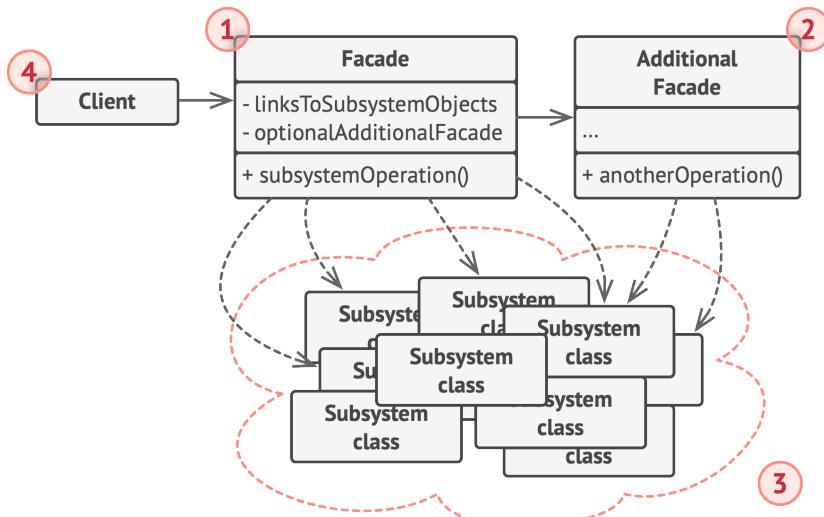
🚗 실제상황 적용



전화로 주문하기.

전화로 주문하기 위해 매장에 전화를 걸었을 때 전화를 받는 교환원이 바로 상점의 모든 서비스와 부서에 대한 당신의 퍼사드입니다. 이때 교환원은 주문 시스템, 지불 게이트웨이 및 다양한 배송 서비스에 대한 간단한 음성 인터페이스를 제공합니다.

구조



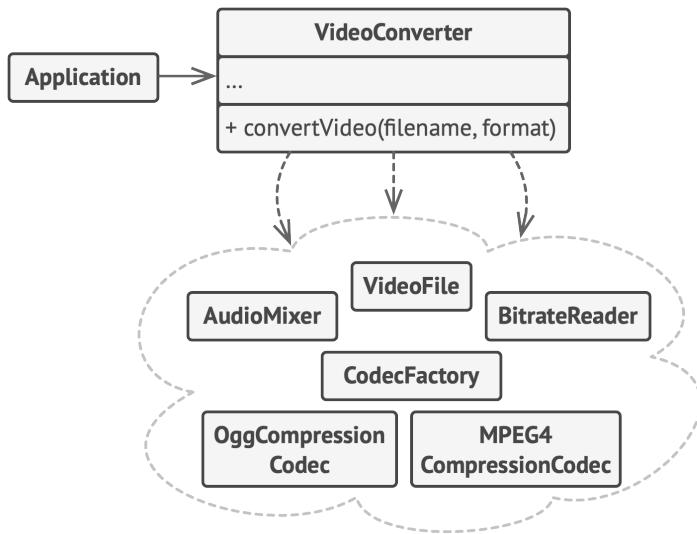
1. **퍼사드** 패턴을 사용하면 하위 시스템 기능들의 특정 부분에 편리하게 접근할 수 있습니다. 또 퍼사드는 클라이언트의 요청을 어디로 보내야 하는지와 움직이는 모든 부품을 어떻게 작동해야 하는지를 알고 있습니다.
2. **추가적인 퍼사드** 클래스를 생성하여 하나의 퍼사드를 관련 없는 기능들로 오염시켜 복잡한 구조로 만드는 것을 방지할 수 있습니다. 추가 퍼사드들은 클라이언트들과 다른 퍼사드들 모두에 사용할 수 있습니다.
3. **복잡한 하위 시스템**은 수십 개의 다양한 객체들로 구성됩니다. 이 모든 객체가 의미 있는 작업을 수행하도록 하려면, 하위 시스템의 세부적인 구현 정보를 깊이 있게 살펴야 합니다. 예를 들어 올바른

순서로 객체들을 초기화하고 그들에게 적절한 형식의 데이터를 제공하는 등의 작업을 수행해야 합니다.

하위 시스템 클래스들은 퍼사드의 존재를 인식하지 못합니다. 이들은 시스템 내에서 작동하며, 매개체 없이 직접 서로와 작업합니다.

4. **클라이언트**는 하위 시스템 객체들을 직접 호출하는 대신 퍼사드를 사용합니다.

의사코드



단일 퍼사드 클래스 내에서 여러 종속 관계들을 격리하는 예시.

이 예시에서 **퍼사드** 패턴은 복잡한 비디오 변환 프레임워크와의 상호작용을 단순화합니다.

당신의 코드가 수십 개의 프레임워크 클래스들과 직접 작업하도록 하는 대신, 해당 기능들을 캡슐화하여 코드의 나머지 부분으로부터 숨기는 퍼사드 클래스를 만듭니다. 이러한 구조를 이용하면 나중에 프레임워크를 업그레이드하거나 다른 프레임워크로 교체할 때 들어갈 노력을 최소한으로 줄일 수 있습니다. 앱에서 바꾸어야 할 부분이 퍼사드의 메서드들의 구현뿐일 것이기 때문입니다.

```
1 // 이것들은 복잡한 타사 비디오 변환 프레임워크 클래스의 일부입니다. 해당 프레임워크
2 // 코드는 우리가 제어할 수 없기 때문에 단순화할 수 없습니다.
3
4 class VideoFile
5 // ...
6
7 class OggCompressionCodec
8 // ...
9
10 class MPEG4CompressionCodec
11 // ...
12
13 class CodecFactory
14 // ...
15
16 class BitrateReader
17 // ...
```

```
19 class AudioMixer
20 // ...
21
22
23 // 퍼사드 클래스를 만들어 프레임워크의 복잡성을 간단한 인터페이스 뒤에 숨길 수
24 // 있습니다. 기능성과 단순함을 상호보완하는 것이죠.
25 class VideoConverter is
26 method convert(filename, format):File is
27     file = new VideoFile(filename)
28     sourceCodec = (new CodecFactory).extract(file)
29     if (format == "mp4")
30         destinationCodec = new MPEG4CompressionCodec()
31     else
32         destinationCodec = new OggCompressionCodec()
33     buffer = BitrateReader.read(filename, sourceCodec)
34     result = BitrateReader.convert(buffer, destinationCodec)
35     result = (new AudioMixer()).fix(result)
36     return new File(result)
37
38 // 애플리케이션 클래스들은 복잡한 프레임워크에서 제공하는 수많은 클래스에 의존하지
39 // 않습니다. 또한 프레임워크의 전환을 결정한 경우에는 퍼사드 클래스만 다시 작성하면
40 // 됩니다.
41 class Application is
42 method main() is
43     convertor = new VideoConverter()
44     mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
45     mp4.save()
```

💡 적용

💡 퍼사드 패턴은 당신이 복잡한 하위 시스템에 대한 제한적이지만 간단한 인터페이스가 필요할 때 사용하세요.

👉 하위 시스템은 시간이 지날수록 더 복잡해지곤 합니다. 디자인 패턴들을 적용하더라도 보통은 생성되는 클래스들이 점점 더 많아지게 됩니다. 하위 시스템은 더 유연해지고 더 많은 다양한 상황에서 재사용할 수 있도록 변경될 수도 있지만, 해당 시스템이 클라이언트에게 요구하는 설정 및 상용구 코드의 양은 점점 더 많아질 것입니다. 이 문제를 해결하기 위해 퍼사드는 대부분의 클라이언트 요구에 부합하면서 하위 시스템에서 가장 많이 사용되는 기능들로 향하는 지름길을 제공합니다.

💡 퍼사드 패턴은 하위 시스템을 계층들로 구성하려는 경우 사용하세요.

👉 하위시스템의 각 계층에 대한 진입점을 정의하기 위해 퍼사드 패턴들을 생성하세요. 당신은 여러 하위시스템이 퍼사드 패턴들을 통해서만 통신하도록 함으로써 해당 하위시스템 간의 결합도를 줄일 수 있습니다.

예를 들어 당신의 비디오 변환 프레임워크는 비디오 또는 오디오 관련의 두 가지 계층으로 나뉠 수 있습니다. 각 계층에 대해 퍼사드를 만든 다음 각 계층의 클래스들이 해당 퍼사드들을 통해

서로 통신하도록 할 수 있습니다. 이러한 접근 방식은 **중재자** 패턴과 매우 유사합니다.

▣ 구현방법

1. 기존 하위시스템이 이미 제공하고 있는 것보다 더 간단한 인터페이스를 제공하는 것이 가능한지 확인하세요. 이 인터페이스가 클라이언트 코드를 하위시스템의 여러 클래스로부터 독립시킨다면 제대로 하고 계신 겁니다.
2. 새 퍼사드 패턴 클래스에서 이 인터페이스를 선언하고 구현하세요. 이 퍼사드는 클라이언트 코드의 호출들을 하위 시스템의 적절한 객체들로 리다이렉션해야 합니다. 이 퍼사드는 클라이언트가 아래 작업을 이미 수행하지 않는 한 하위시스템을 초기화하고 추가 수명 주기를 관리하는 작업의 책임을 맡아야 합니다.
3. 패턴을 최대한 활용하려면 모든 클라이언트 코드가 퍼사드 패턴을 통해서만 하위시스템과 통신하도록 하세요. 그러면 이제 클라이언트 코드는 하위시스템 코드의 변경 사항들로부터 보호됩니다. 예를 들어, 하위시스템이 새 버전으로 업그레이드되면 당신은 퍼사드의 코드만 수정하면 됩니다.
4. 퍼사드가 **너무 커지면** 행동들 일부를 새롭고 정제된 퍼사드 클래스로 추출하는 것을 고려하세요.

⚠️ 장단점

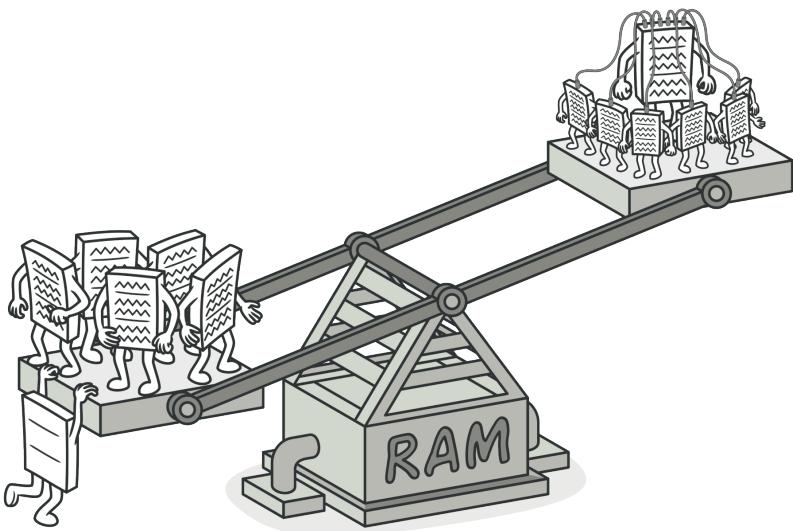
- ✓ 복잡한 하위 시스템에서 코드를 별도로 분리할 수 있습니다.
- ✗ 퍼사드는 앱의 모든 클래스에 결합된 전지전능한 객체가 될 수 있습니다.

↔ 다른 패턴과의 관계

- 퍼사드는 기존 객체들을 위한 새 인터페이스를 정의하는 반면 어댑터는 기존의 인터페이스를 사용할 수 있게 만들려고 노력합니다. 또 어댑터는 일반적으로 하나의 객체만 래핑하는 반면 퍼사드는 많은 객체의 하위시스템과 함께 작동합니다.
- 추상 팩토리는 하위시스템 객체들이 클라이언트 코드에서 생성되는 방식만 숨기고 싶을 때 퍼사드 대신 사용할 수 있습니다.
- 플라이웨이트는 작은 객체들을 많이 만드는 방법을 보여 주는 반면 퍼사드 패턴은 전체 하위 시스템을 나타내는 단일 객체를 만드는 방법을 보여 줍니다.
- 중재자와 퍼사드 패턴은 비슷한 역할을 합니다. 둘 다 밀접하게 결합된 많은 클래스 간의 협업을 구성하려고 합니다.
 - 퍼사드 패턴은 객체들의 하위 시스템에 대한 단순화된 인터페이스를 정의하지만 새로운 기능을 도입하지는 않습니다.

하위 시스템 자체는 퍼사드를 인식하지 못하며, 하위 시스템 내의 객체들은 서로 직접 통신할 수 있습니다.

- 중재자는 시스템 컴포넌트 간의 통신을 중앙 집중화합니다. 컴포넌트들은 중재자 객체에 대해서만 알며 서로 직접 통신하지 않습니다.
- 대부분의 경우 하나의 퍼사드 객체만 있어도 충분하므로 퍼사드 패턴의 클래스는 종종 싱글턴으로 변환될 수 있습니다.
- 퍼사드 패턴은 복잡한 객체 또는 시스템을 보호하고 자체적으로 초기화한다는 점에서 프록시와 유사합니다. 퍼사드 패턴과 달리 프록시는 자신의 서비스 객체와 같은 인터페이스를 가지므로 이들은 서로 상호 교환이 가능합니다.



플라이웨이트 패턴

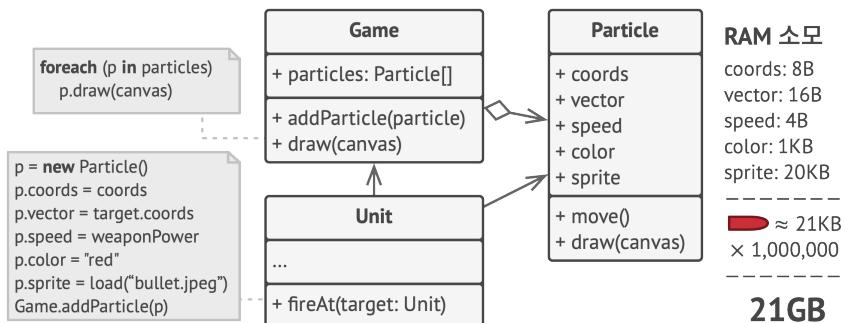
다음 이름으로도 불립니다: 캐시, Flyweight

플라이웨이트는 각 객체에 모든 데이터를 유지하는 대신 여러 객체들 간에 상태의 공통 부분들을 공유하여 사용할 수 있는 RAM에 더 많은 객체들을 포함할 수 있도록 하는 구조 디자인 패턴입니다.

(?) 문제

당신은 재미 삼아 플레이어들이 지도를 돌아다니며 서로에게 총을 쏘는 간단한 비디오 게임을 만들기로 했습니다. 당신은 폭발들로 인한 방대한 양의 총알들, 미사일들 및 파편들이 지도 전체를 날아다니는 전율 넘치는 경험을 플레이어들에게 선사하기로 했으며, 이를 선사하기 위해 사실적인 입자 시스템을 구현하기로 했습니다.

당신은 게임을 완성한 후 친구에게 게임을 보냈습니다. 당신의 컴퓨터에서는 게임이 완벽하게 실행되었지만, 당신의 친구는 오랫동안 게임을 즐길 수 없었습니다. 왜냐하면 친구의 컴퓨터에서는 시작 후 고작 몇 분 후에 게임이 계속 충돌했기 때문입니다. 당신이 디버그 로그를 자세히 살펴본 결과, 친구의 컴퓨터의 RAM이 당신의 컴퓨터처럼 충분하지 않아 게임이 충돌했음이 분명해졌습니다.

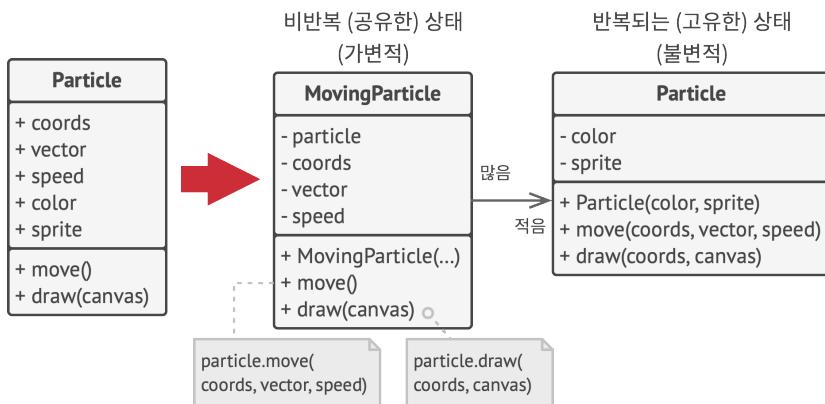


문제의 원인은 당신의 입자 시스템과 관련이 있었습니다. 각 총알, 미사일 또는 파편과 같은 입자는 많은 데이터를 포함하는 별도의

객체로 표시되었습니다. 플레이어 화면의 대학살이 절정에 이르렀을 때 새로 생성된 입자들을 더 이상 나머지 RAM이 감당하지 못해서 프로그램이 충돌했습니다.

😊 해결책

`Particle` (입자) 클래스를 자세히 살펴보면 `color` (색상) 및 `sprite` (스프라이트) 필드들이 다른 필드들보다 훨씬 더 많은 메모리를 사용한다는 것을 알 수 있습니다. 더 나쁜 것은 이 두 필드가 모든 입자에 걸쳐 거의 같은 데이터를 저장한다는 것입니다. 예를 들어, 모든 총알은 같은 색상과 스프라이트를 갖습니다.

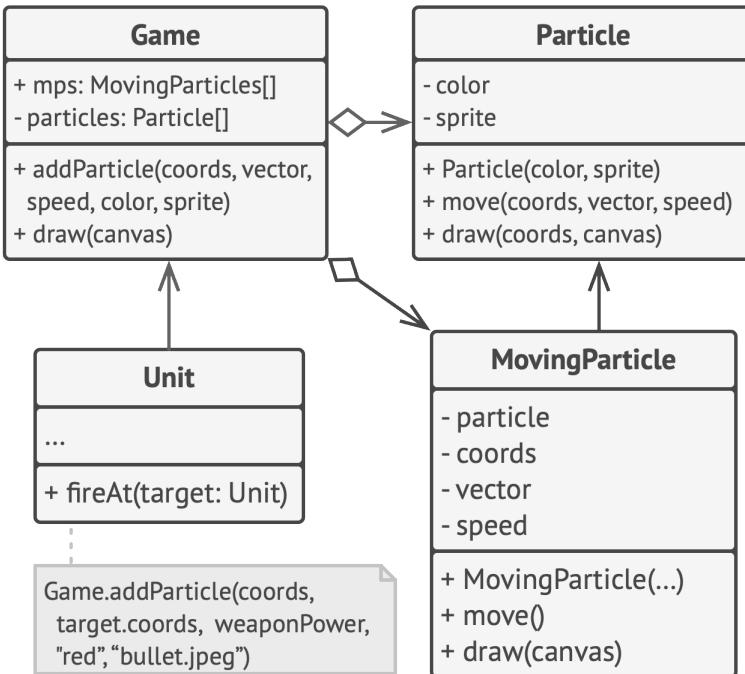


좌표, 이동 벡터 및 속도와 같은 입자 상태의 다른 부분들은 각 입자마다 고유하며, 이러한 필드들의 값은 시간이 지남에 따라 변경됩니다. 이 데이터는 입자의 계속 변화하는 콘텍스트를

나타내나, 반면 색상과 스프라이트는 각 입자마다 일정하게 유지됩니다.

객체의 이러한 상수 데이터를 일반적으로 고유한 상태라고 합니다. 이 데이터는 객체 안에서 삽니다. 다른 객체들은 이 데이터를 읽을 수만 있고 변경할 수는 없습니다. 종종 다른 객체들에 의해 '외부에서' 변경되는 객체의 나머지 상태를 공유한 상태라고 합니다.

플라이웨이트 패턴은 객체 내부에 공유한 상태의 저장을 중단하고, 대신 이 상태를 이 상태에 의존하는 특정 메서드들에 전달할 것을 제안합니다. 고유한 상태만 객체 내에 유지되므로 해당 고유한 상태는 콘텍스트가 다른 곳에서 재사용할 수 있습니다. 이러한 객체들은 공유한 상태보다 변형이 훨씬 적은 고유한 상태에서만 달라지므로 훨씬 더 적은 수의 객체만 있으면 됩니다.



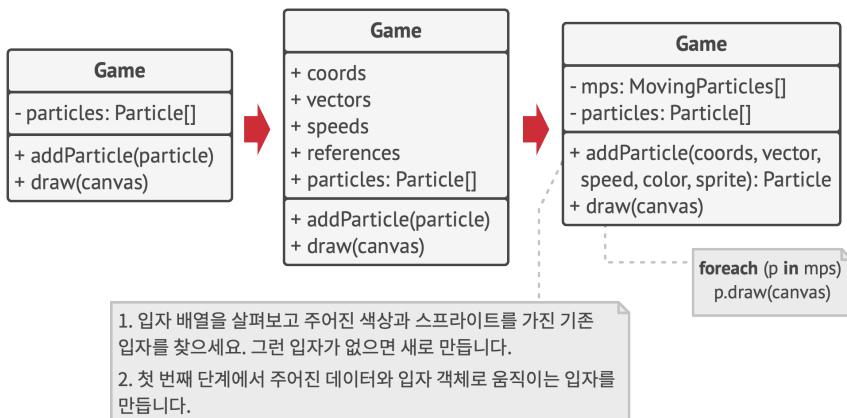
RAM 소모	coords: 8B	color: 4B	vector: 16B	particle: 4B	speed: 4B	× 1	× 1,000,000	32MB
		sprite: 20KB				× 1	× 1,000,000	
		-----	-----					
		≈ 21KB						
					≈ 32B			

이제 당신의 게임을 다시 살펴봅시다. 입자 클래스에서 공유한 상태를 추출했다고 가정하면 총알, 미사일, 파편의 세 가지 다른 객체만으로도 게임의 모든 입자를 충분히 나타낼 수 있습니다. 지금쯤 짐작하셨겠지만 고유한 상태만 저장하는 객체를 플라이웨이트라고 합니다.

공유한 상태 스토리지

공유한 상태는 어디로 이동할까요? 일부 클래스가 이 상태를 여전히 저장하고 있는 거겠죠? 대부분의 경우 공유한 상태는 패턴을 적용하기 전에 객체들을 집합시키는 컨테이너 객체로 이동됩니다.

당신의 게임에서 이것은 `particle` 필드에 모든 입자를 저장하는 주요 `Game` 객체입니다. 공유한 상태를 이 클래스로 이동하려면 개별 입자의 좌표, 벡터 및 속도를 저장하기 위한 여러 배열 필드들을 생성해야 합니다. 거기서 끝이 아닙니다. 입자를 나타내는 특정 플라이웨이트에 대한 참조를 저장하려면 다른 배열이 필요합니다. 이러한 배열들은 같은 인덱스를 사용하여 입자의 모든 데이터에 액세스할 수 있도록 동기화되어야 합니다.



이보다 더 훌륭한 해결책은 플라이웨이트 객체에 대한 참조와 함께 공유된 상태를 저장할 별도의 콘텍스트 클래스를 만드는

것입니다. 이 접근 방식을 사용하려면 컨테이너 클래스에 단일 배열만 있으면 됩니다.

잠시만요! 처음에 그랬던 것처럼 이런 콘텍스트 객체들이 많이 있어야 하지 않나요? 맞습니다. 그러나 이제는 이러한 객체들이 이전보다 훨씬 작습니다. 가장 메모리를 많이 사용하는 필드들이 고작 몇 개의 플라이웨이트 객체들로 이동되었습니다. 이제 하나의 커다란 플라이웨이트 객체를 몇천 개의 작은 콘텍스트 객체들이 재사용할 수 있으며, 더 이상 커다란 플라이웨이트 객체의 데이터의 천 개의 복사본을 저장할 필요가 없습니다.

플라이웨이트와 불변성

같은 플라이웨이트 객체가 다른 콘텍스트들에서 사용될 수 있으므로 해당 플라이웨이트 객체의 상태를 수정할 수 없는지 확인해야 합니다. 플라이웨이트는 생성자 매개변수들을 통해 상태를 한 번만 초기화해야 합니다. 또 setter 또는 public 필드들을 다른 객체들에 노출해서는 안 됩니다.

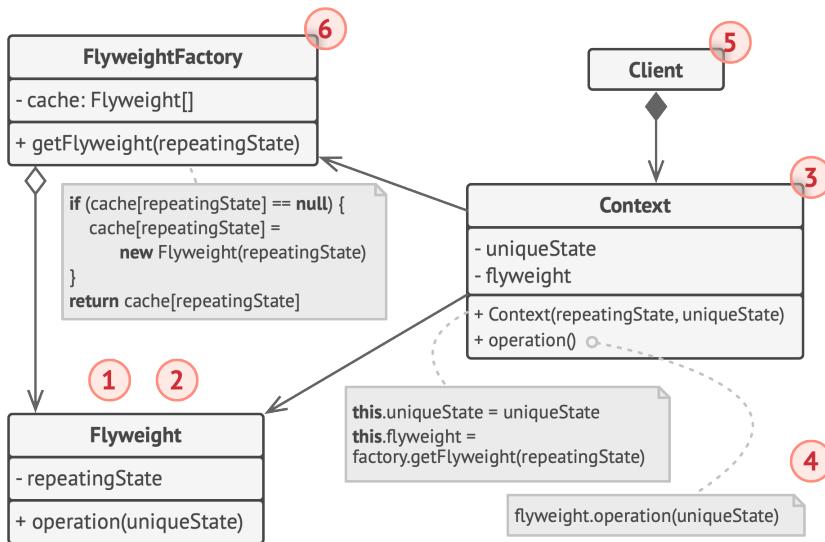
플라이웨이트 팩토리

다양한 플라이웨이트들에 보다 편리하게 액세스하기 위해 기존 플라이웨이트 객체들의 풀을 관리하는 팩토리 메서드를 생성할 수 있습니다. 이 메서드는 클라이언트에서 원하는 플라이웨이트의 고유한 상태를 받아들이고 이 상태와 일치하는 기존 플라이웨이트

객체를 찾고 발견되면 반환합니다. 그렇지 않으면 새 플라이웨이트를 생성하여 풀에 추가합니다.

이 메서드를 배치할 수 있는 몇 가지 옵션이 있습니다. 그중 가장 확실한 장소는 플라이웨이트 컨테이너입니다. 대안으로 당신은 새 팩토리 클래스를 생성할 수 있고, 또 팩토리 메서드를 정적으로 만들고 실제 플라이웨이트 클래스에 넣을 수 있습니다.

구조



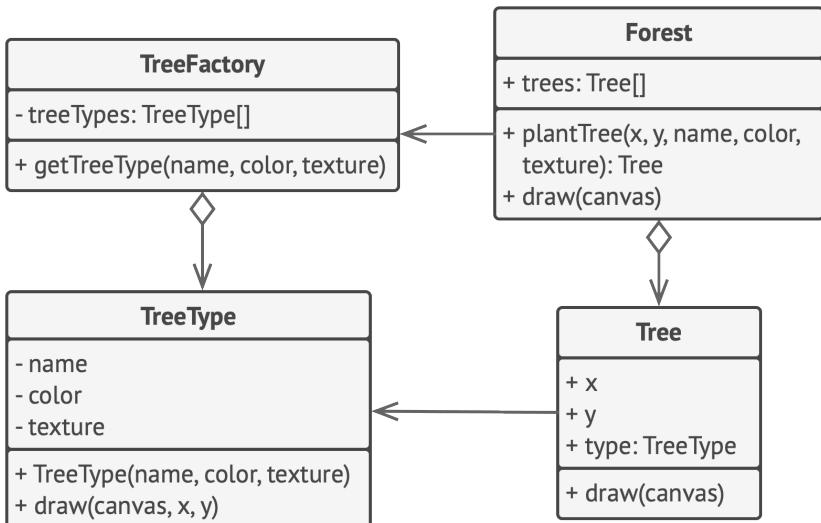
1. 플라이웨이트 패턴은 단지 최적화에 불과합니다. 이 패턴을 적용하기 전에 프로그램이 동시에 메모리에 유사한 객체들을 대량으로 보유하는 것과 관련된 RAM 소비 문제가 있는지 확인하시고 이 문제가 다른 의미 있는 방법으로 해결될 수 없는지도 확인하세요.

2. **플라이웨이트** 클래스에는 여러 객체들 간에 공유할 수 있는 원래 객체의 상태의 부분이 포함됩니다. 같은 플라이웨이트 객체가 다양한 콘텍스트에서 사용될 수 있습니다. 플라이웨이트 내부에 저장된 상태를 고유한(intrinsic) 상태라고 하며, 플라이웨이트의 메서드에 전달된 상태를 공유한(extrinsic) 상태라고 합니다.
3. **콘텍스트** 클래스는 공유한 상태를 포함하며, 이 상태는 모든 원본 객체들에서 고유합니다. 콘텍스트가 플라이웨이트 객체 중 하나와 쌍을 이루면 원래 객체의 전체 상태를 나타냅니다.
4. 일반적으로 원래 객체의 행동은 플라이웨이트 클래스에 남아 있습니다. 이 경우 플라이웨이트의 메서드의 호출자는 공유한 상태의 적절한 부분들을 메서드의 매개변수들에 전달해야 합니다. 반면에, 행동은 콘텍스트 클래스로 이동할 수 있으며, 이 클래스는 연결된 플라이웨이트를 단순히 데이터 객체로 사용할 것입니다.
5. **클라이언트**는 플라이웨이트들의 공유된 상태를 저장하거나 계산합니다. 클라이언트의 관점에서 플라이웨이트는 일부 콘텍스트 데이터를 그의 메서드들의 매개변수들에 전달하여 런타임에 설정될 수 있는 템플릿 객체입니다.
6. **플라이웨이트 팩토리**는 기존 플라이웨이트들의 풀을 관리합니다. 이 팩토리로 인해 클라이언트들은 플라이웨이트들을 직접 만들지 않는 대신 원하는 플라이웨이트의 고유한 상태의 일부를 전달하여 공장을 호출합니다. 팩토리는 이전에 생성된 플라이웨이트들을

살펴보고 검색 기준과 일치하는 기존 플라이웨이트를 반환하거나 기준에 맞는 플라이웨이트가 발견되지 않으면 새로 생성합니다.

의사코드

이 예시에서 **플라이웨이트** 패턴은 캔버스에 수백만 개의 나무 객체들을 렌더링할 때 메모리 사용량을 줄이는 데 도움을 줍니다.



이 패턴은 주요 **Tree** (나무) 클래스에서 반복되는 고유한 상태를 추출하여 **TreeType** (나무 종류) 플라이웨이트 클래스로 이동합니다.

같은 데이터를 여러 객체에 저장하는 대신 이제 몇 개의 플라이웨이트 객체들에 보관되고 콘텍스트 역할을 하는 적절한 **Tree** 객체들에 연결됩니다. 클라이언트 코드는 플라이웨이트

팩토리를 사용하여 새 Tree 객체들을 생성합니다. 이 팩토리는 올바른 객체를 검색하고 필요한 경우 재사용하는 작업의 복잡성을 캡슐화합니다.

```

1 // 플라이웨이트 클래스는 트리의 상태 일부를 포함합니다. 이러한 필드는 각 특정
2 // 트리에 대해 고유한 값을 저장합니다. 예를 들어 여기에서는 트리 좌표들을 찾을
3 // 수 없을 것입니다. 그러나 많은 트리들이 공유하는 질감들과 색상들은 찾을 수 있을
4 // 것입니다. 이 데이터는 일반적으로 크기 때문에 각 트리 개체에 보관하면 많은
5 // 메모리를 낭비하게 됩니다. 대신 질감, 색상 및 기타 반복되는 데이터를 많은 개별
6 // 트리 객체들이 참조할 수 있는 별도의 객체로 추출할 수 있습니다.
7 class TreeType is
8   field name
9   field color
10  field texture
11  constructor TreeType(name, color, texture) { ... }
12  method draw(canvas, x, y) is
13    // 1. 주어진 유형, 색상 및 질감의 비트맵을 만드세요.
14    // 2. 비트맵을 캔버스의 X 및 Y 좌표에 그리세요.
15
16 // 플라이웨이트 팩토리는 기존 플라이웨이트를 재사용할지 아니면 새로운 객체를
17 // 생성할지를 결정합니다.
18 class TreeFactory is
19   static field treeTypes: collection of tree types
20   static method getTreeType(name, color, texture) is
21     type = treeTypes.find(name, color, texture)
22     if (type == null)
23       type = new TreeType(name, color, texture)
24     treeTypes.add(type)
25   return type
26
27 // 콘텍스트 객체는 트리 상태의 공유된 부분을 포함합니다. 이러한 부분들은 두 개의

```

```
28 // 정수로 된 좌표와 하나의 참조 필드만 참조하여 크기가 작기 때문에 하나의 앱이
29 // 이런 부분을 수십억 개씩 만들 수 있습니다.
30 class Tree is
31     field x,y
32     field type: TreeType
33     constructor Tree(x, y, type) { ... }
34     method draw(canvas) is
35         type.draw(canvas, this.x, this.y)
36
37 // Tree 및 Forest 클래스들은 플라이웨이트의 클라이언트들이며 Tree 클래스를 더
38 // 이상 개발할 계획이 없으면 이 둘을 병합할 수 있습니다.
39 class Forest is
40     field trees: collection of Trees
41
42     method plantTree(x, y, name, color, texture) is
43         type = TreeFactory.getType(name, color, texture)
44         tree = new Tree(x, y, type)
45         trees.add(tree)
46
47     method draw(canvas) is
48         foreach (tree in trees) do
49             tree.draw(canvas)
```

💡 적용

💡 **플라이웨이트 패턴은 당신의 프로그램이 많은 수의 객체들을 지원해야 해서 사용할 수 있는 RAM을 거의 다 사용했을 때만 사용하세요.**

⚡ 이 패턴 적용의 혜택은 패턴을 사용하는 방법과 위치에 따라 크게 달라지며, 다음과 같은 경우에 가장 유용합니다.

- 앱이 수많은 유사 객체들을 생성해야 할 때
- 이것이 대상 장치에서 사용할 수 있는 모든 RAM을 소모할 때
- 이 객체들에 여러 중복 상태들이 포함되어 있으며, 이 상태들이 추출된 후 객체 간에 공유될 수 있을 때

▣ 구현방법

1. 플라이웨이트가 될 클래스의 필드들을 두 부분으로 나누세요.
 - 고유한 상태: 많은 객체에 걸쳐 복제된 변경되지 않는 데이터를 포함하는 필드들
 - 공유한 상태: 각 객체에 고유한 콘텍스트 데이터를 포함하는 필드들

2. 클래스의 고유한 상태를 나타내는 필드들은 그대로 두되 변경될 수 없도록 하세요. 이 필드들은 생성자 내부에서만 초기값들을 가져와야 합니다.
3. 공유한 상태의 필드들을 사용하는 메서드들을 살펴보세요. 메서드에 사용된 각 필드에 대해 새 매개변수를 도입하고 필드 대신 사용하세요.
4. 옵션으로, 플라이웨이트들의 풀을 관리하기 위한 팩토리 클래스를 생성하세요. 이 클래스는 새 플라이웨이트를 만들기 전에 기존 플라이웨이트의 존재 여부를 확인해야 합니다. 팩토리가 설치되면 고객은 팩토리를 통해서만 플라이웨이트를 요청해야 합니다. 그들은 팩토리에 플라이웨이트의 고유한 상태를 전달하여 원하는 플라이웨이트를 설명해야 합니다.
5. 클라이언트는 플라이웨이트 객체들의 메서드들을 호출할 수 있도록 공유한 상태의 값들(콘텍스트)을 저장하거나 계산해야 합니다. 편의상, 플라이웨이트를 참조하는 필드와 공유한 상태는 별도의 콘텍스트 클래스로 이동할 수 있습니다.

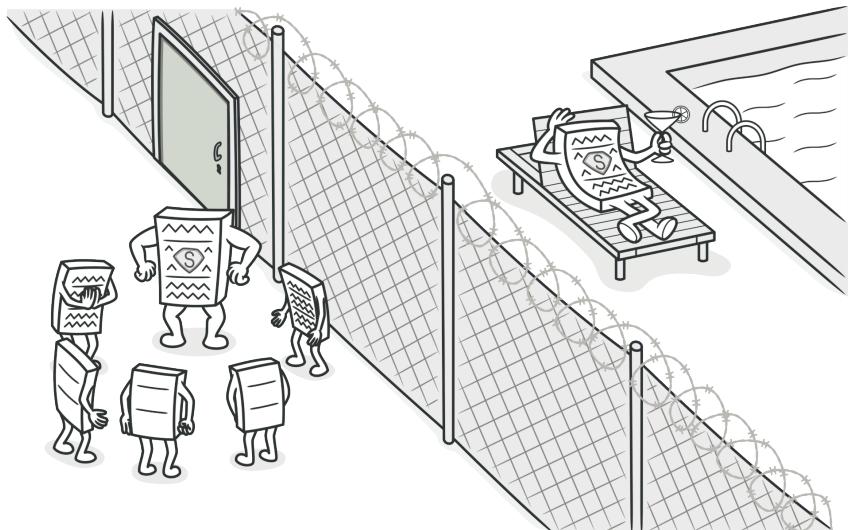
⚠️ 장단점

- ✓ 당신의 프로그램에 유사한 객체들이 많다고 가정하면 많은 RAM을 절약할 수 있습니다.

- ✖ 누군가가 플라이웨이트 메서드를 호출할 때마다 콘텍스트 데이터의 일부를 다시 계산해야 한다면 당신은 CPU 주기 대신 RAM을 절약하고 있는 것일지도 모릅니다.
- ✖ 코드가 복잡해지므로 새로운 팀원들은 왜 개체(entity)의 상태가 그런 식으로 분리되었는지 항상 궁금해할 것입니다.

↔ 다른 패턴과의 관계

- RAM을 절약하기 위하여 **복합체** 패턴 트리의 공유된 잎 노드들을 **플라이웨이트들**로 구현할 수 있습니다.
- **플라이웨이트**는 작은 객체들을 많이 만드는 방법을 보여 주는 반면 **퍼사드** 패턴은 전체 하위 시스템을 나타내는 단일 객체를 만드는 방법을 보여 줍니다.
- 만약 객체들의 공유된 상태들을 단 하나의 플라이웨이트 객체로 줄일 수 있다면 **플라이웨이트는 싱글턴**과 유사해질 수 있습니다. 그러나 이 패턴들에는 두 가지 근본적인 차이점이 있습니다:
 1. 싱글턴은 인스턴스가 하나만 있어야 합니다. 반면에 **플라이웨이트** 클래스는 여러 고유한 상태를 가진 여러 인스턴스를 포함할 수 있습니다.
 2. 싱글턴 객체는 변할 수 있습니다 (mutable). 플라이웨이트 객체들은 변할 수 없습니다 (immutable).



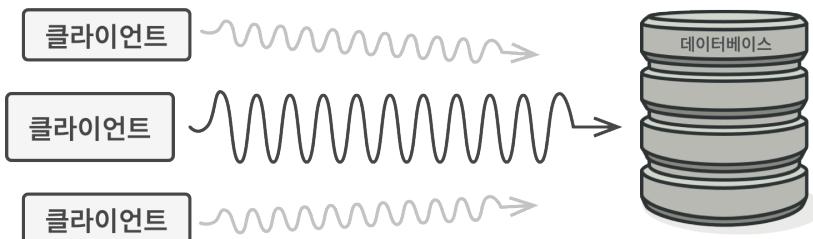
프록시 패턴

다음 이름으로도 불립니다: Proxy

프록시는 다른 객체에 대한 대체 또는 자리표시자를 제공할 수 있는 구조 디자인 패턴입니다. 프록시는 원래 객체에 대한 접근을 제어하므로, 당신의 요청이 원래 객체에 전달되기 전 또는 후에 무언가를 수행할 수 있도록 합니다.

:(문제

객체에 대한 접근을 제한하는 이유는 무엇일까요? 이 질문에 답하기 위하여 방대한 양의 시스템 자원을 소비하는 거대한 객체가 있다고 가정합시다. 이 객체는 필요할 때가 있기는 하지만, 항상 필요한 것은 아닙니다.



데이터베이스 쿼리들은 정말 느릴 수 있습니다.

당신은 실제로 필요할 때만 이 객체를 만들어서 지연된 초기화를 구현할 수 있습니다. 그러면 객체의 모든 클라이언트들은 어떤 지연된 초기화 코드를 실행해야 하는데, 불행히도 이것은 아마도 많은 코드 중복을 초래할 것입니다.

이상적인 상황에서는 이 코드를 객체의 클래스에 직접 넣을 수 있겠지만, 그게 항상 가능한 것은 아닙니다. 예를 들어 그 클래스가 폐쇄된 타사 라이브러리의 일부일 수 있습니다.

😊 해결책

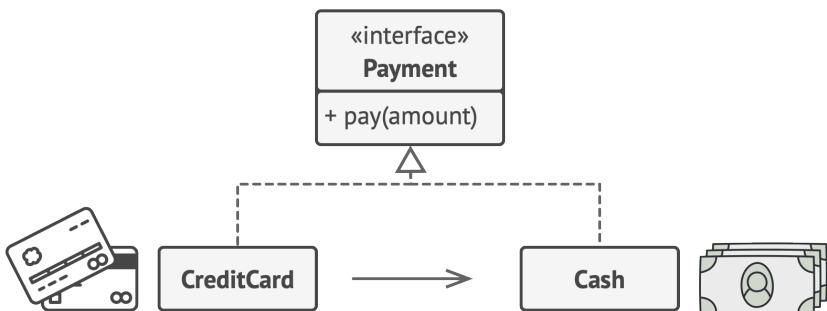
프록시 패턴은 원래 서비스 객체와 같은 인터페이스로 새 프록시 클래스를 생성하라고 제안합니다. 그러면 프록시 객체를 원래 객체의 모든 클라이언트들에 전달하도록 앱을 업데이트할 수 있습니다. 클라이언트로부터 요청을 받으면 이 프록시는 실제 서비스 객체를 생성하고 모든 작업을 이 객체에 위임합니다.



프록시는 데이터베이스 객체로 자신을 변장합니다. 프록시는 지연된 초기화 및 결값 캐싱을 클라이언트와 실제 데이터베이스 객체가 알지 못하는 상태에서 처리할 수 있습니다.

그러나 이것들은 무슨 소용이 있는지 궁금하실 것입니다. 당신이 클래스의 메인 로직 이전이나 이후에 무언가를 실행해야 하는 경우 프록시는 해당 클래스를 변경하지 않고도 이 무언가를 수행할 수 있도록 합니다. 프록시는 원래 클래스와 같은 인터페이스를 구현하므로 실제 서비스 객체를 기대하는 모든 클라이언트에 전달될 수 있습니다.

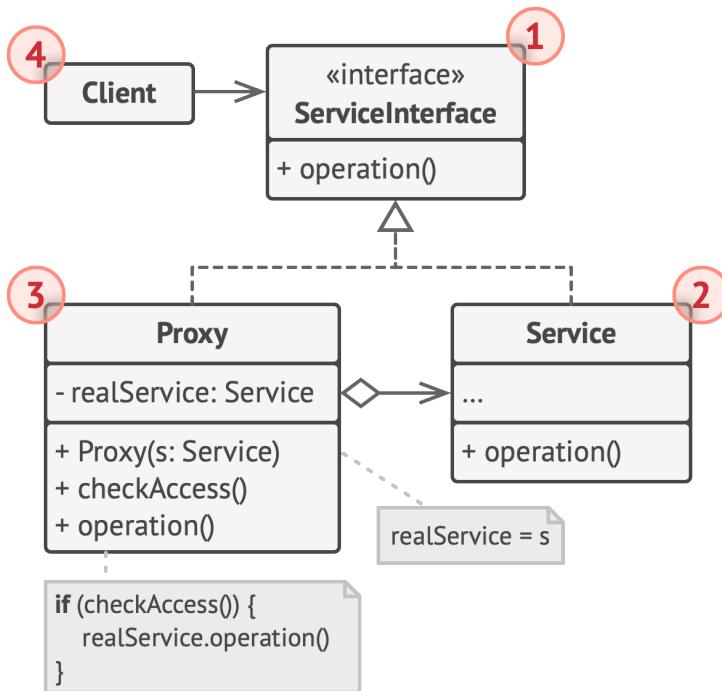
🚗 실제상황 적용



신용 카드는 현금과 마찬가지로 결제에 사용할 수 있습니다.

신용 카드는 은행 계좌의 프록시이며, 은행 계좌는 현금의 프록시입니다. 둘 다 같은 인터페이스를 구현하며 둘 다 결제에 사용될 수 있습니다. 신용 카드를 사용하는 소비자는 많은 현금을 가지고 다닐 필요가 없어서 기분이 좋습니다. 또한 상점 주인은 거래 수입을 은행에 가는 길에 강도를 당하거나 잃어버릴 위험 없이 계좌에 전자적으로 입금이 되기 때문에 기분이 좋습니다.

구조



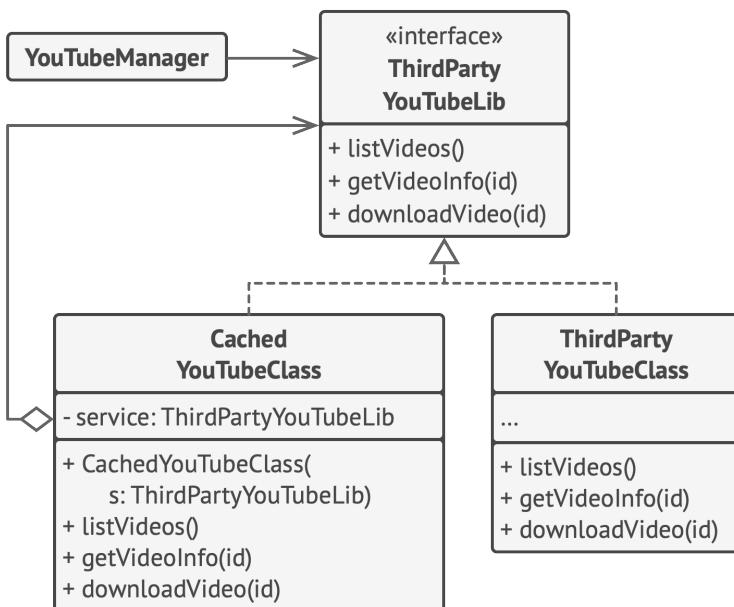
1. **서비스 인터페이스**는 서비스의 인터페이스를 선언합니다. 프록시가 서비스 객체로 위장할 수 있으려면 이 인터페이스를 따라야 합니다.
2. **서비스**는 어떤 유용한 비즈니스 로직을 제공하는 클래스입니다.
3. **프록시** 클래스에는 서비스 객체를 가리키는 참조 필드가 있습니다. 프록시가 요청의 처리(예: 초기화 지연, 로깅, 액세스 제어, 캐싱 등)를 완료하면, 그 후 처리된 요청을 서비스 객체에 전달합니다.

일반적으로 프록시들은 서비스 객체들의 전체 수명 주기를 관리합니다.

- 클라이언트는 같은 인터페이스를 통해 서비스들 및 프록시들과 함께 작동해야 합니다. 그러면 서비스 객체를 기대하는 모든 코드에 프록시를 전달할 수 있기 때문입니다.

의사코드

이 예시는 **프록시** 패턴이 제삼자 유튜브 통합 라이브러리에 지연된 초기화 및 캐싱을 도입하는 데 어떻게 도움이 되는지 보여줍니다.



프록시를 사용하여 서비스의 결과를 캐싱합니다.

이 라이브러리는 비디오 다운로드 클래스를 제공하나 매우 비효율적입니다. 왜냐하면 클라이언트 앱이 같은 비디오를 여러 번 요청하면 라이브러리는 처음 다운로드한 파일을 캐싱하고 재사용하는 대신 계속해서 같은 비디오를 다운로드하기 때문입니다.

프록시 클래스는 원래 다운로더와 같은 인터페이스를 구현하고 이 다운로더에 모든 작업을 위임하나, 앱이 같은 비디오를 두 번 이상 요청하면 이미 다운로드한 파일을 추적한 후 캐시 된 결과를 반환합니다.

```

1 // 원격 서비스의 인터페이스.
2 interface ThirdPartyYouTubeLib is
3     method listVideos()
4     method getVideoInfo(id)
5     method downloadVideo(id)
6
7 // 서비스 연결자의 구상 구현. 이 클래스의 메서드들은 유튜브에서 정보를 요청할 수
8 // 있습니다. 해당 요청의 속도는 사용자와 유튜브의 인터넷 연결 속도에 따라 다를
9 // 것입니다. 앱이 많은 요청을 동시에 처리하면 속도가 느려질 것입니다. 이는
10 // 요청들이 모두 같은 정보를 요청하더라도 마찬가지입니다.
11 class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib is
12     method listVideos() is
13         // 유튜브에 API 요청을 보냅니다.
14
15     method getVideoInfo(id) is
16         // 어떤 비디오에 대한 메타데이터를 가져옵니다.
17
18     method downloadVideo(id) is

```

```
19     // 유튜브에서 동영상 파일을 다운로드합니다.
20
21 // 일부 대역폭을 절약하기 위해 요청 결과를 캐시하고 일정 기간 보관할 수 있습니다.
22 // 그러나 이러한 코드를 서비스 클래스에 직접 넣는 것은 불가능할 수 있습니다. 예를
23 // 들어, 타사 라이브러리의 일부로 제공되었거나 `final`로 정의된 경우에는 말이죠.
24 // 서비스 클래스와 같은 인터페이스를 구현하는 새 프록시 클래스에 캐싱 코드를 넣는
25 // 이유가 바로 그 때문입니다. 이 클래스는 실제 요청을 보내야 하는 경우에만 서비스
26 // 객체에 위임합니다.
27 class CachedYouTubeClass implements ThirdPartyYouTubeLib {
28     private field service: ThirdPartyYouTubeLib
29     private field listCache, videoCache
30     field needReset
31
32     constructor CachedYouTubeClass(service: ThirdPartyYouTubeLib) {
33         this.service = service
34
35     method listVideos() {
36         if (listCache == null || needReset)
37             listCache = service.listVideos()
38
39         return listCache
40
41     method getVideoInfo(id) {
42         if (videoCache == null || needReset)
43             videoCache = service.getVideoInfo(id)
44
45     method downloadVideo(id) {
46         if (!downloadExists(id) || needReset)
47             service.downloadVideo(id)
48
49 // 서비스 객체와 직접 작업하던 그래픽 사용자 인터페이스 클래스는 서비스 객체와
50 // 인터페이스를 통해 작업하는 한 변경되지 않습니다. 둘 다 같은 인터페이스를
```

```

51 // 구현하므로 실제 서비스 객체 대신 프록시 객체를 안전하게 전달할 수 있습니다.
52 class YouTubeManager is
53   protected field service: ThirdPartyYouTubeLib
54
55   constructor YouTubeManager(service: ThirdPartyYouTubeLib) is
56     this.service = service
57
58   method renderVideoPage(id) is
59     info = service.getVideoInfo(id)
60     // 비디오 페이지를 렌더링하세요.
61
62   method renderListPanel() is
63     list = service.listVideos()
64     // 비디오 섬네일 리스트를 렌더링하세요.
65
66   method reactOnUserInput() is
67     renderVideoPage()
68     renderListPanel()
69
70 // 앱은 언제든지 프록시를 설정할 수 있습니다.
71 class Application is
72   method init() is
73     aYouTubeService = new ThirdPartyYouTubeClass()
74     aYouTubeProxy = new CachedYouTubeClass(aYouTubeService)
75     manager = new YouTubeManager(aYouTubeProxy)
76     manager.reactOnUserInput()

```

💡 적용

프록시 패턴을 활용하는 방법들은 수십 가지가 있으며, 패턴이 가장 많이 사용되는 용도들을 살펴보겠습니다.

- ❖ 지연된 초기화(가상 프록시). 이것은 어쩌다 필요한 무거운 서비스 객체가 항상 가동되어 있어 시스템 자원들을 낭비하는 경우입니다.
- ❖ 앱이 시작될 때 객체를 생성하는 대신, 객체 초기화를 실제로 초기화가 필요한 시점까지 지연할 수 있습니다.
- ❖ 접근 제어 (보호 프록시). 당신이 특정 클라이언트들만 서비스 객체를 사용할 수 있도록 하려는 경우에 사용할 수 있습니다. 예를 들어 당신의 객체들이 운영 체제의 중요한 부분이고 클라이언트들이 다양한 실행된 응용 프로그램(악의적인 응용 프로그램 포함)인 경우입니다.
- ❖ 이 프록시는 클라이언트의 자격 증명이 어떤 정해진 기준과 일치하는 경우에만 서비스 객체에 요청을 전달할 수 있습니다.
- ❖ 원격 서비스의 로컬 실행 (원격 프록시). 서비스 객체가 원격 서버에 있는 경우입니다.
- ❖ 이 경우 프록시는 네트워크를 통해 클라이언트 요청을 전달하여 네트워크와의 작업의 모든 복잡한 세부 사항을 처리합니다.
- ❖ 요청들의 로깅(로깅 프록시). 서비스 객체에 대한 요청들의 기록을 유지하려는 경우입니다.

- ⚡ 프록시는 각 요청을 서비스에 전달하기 전에 로깅(기록)할 수 있습니다.
- ⚡ 요청 결과들의 캐싱(캐싱 프록시). 이것은 클라이언트 요청들의 결과들을 캐시하고 이 캐시들의 수명 주기를 관리해야 할 때, 특히 결과들이 상당히 큰 경우에 사용됩니다.
- ⚡ 프록시는 항상 같은 결과를 생성하는 반복 요청들에 대해 캐싱을 구현할 수 있습니다. 프록시는 요청들의 매개변수들을 캐시 키들로 사용할 수 있습니다.
- ⚡ 스마트 참조. 이것은 사용하는 클라이언트들이 없어 거대한 객체를 해제할 수 있어야 할 때 사용됩니다.
- ⚡ 프록시는 서비스 객체 또는 그 결과에 대한 참조를 얻은 클라이언트들을 추적할 수 있습니다. 때때로 프록시는 클라이언트들을 점검하여 클라이언트들이 여전히 활성 상태인지를 확인할 수 있습니다. 클라이언트 리스트가 비어 있으면 프록시는 해당 서비스 객체를 닫고 그에 해당하는 시스템 자원을 확보할 수 있습니다.

또 프록시는 클라이언트가 서비스 객체를 수정했는지도 추적할 수 있으며, 변경되지 않은 객체는 다른 클라이언트들이 재사용할 수 있습니다.

▣ 구현방법

1. 기존 서비스 인터페이스가 없는 경우, 서비스 인터페이스를 하나 생성하여 프록시와 서비스 객체 간의 상호 교환을 가능하게 만드세요. 서비스 클래스에서 인터페이스를 추출하는 것이 항상 가능한 것은 아닙니다. 왜냐하면 그 인터페이스를 사용하려면 서비스의 모든 클라이언트를 변경해야 하기 때문입니다. 대신 프록시를 서비스 클래스의 자식 클래스로 만들 수 있으며, 이렇게 하면 서비스의 인터페이스를 상속하게 할 수 있습니다.
2. 프록시 클래스를 만드세요. 이 클래스에는 서비스에 대한 참조를 저장하기 위한 필드가 있어야 합니다. 일반적으로 프록시들은 서비스들의 전체 수명 주기를 생성하고 관리합니다. 또 드물지만, 클라이언트가 서비스를 프록시의 생성자에 전달하는 방식으로 서비스가 프록시에 전달되기도 합니다.
3. 목적에 따라 프록시 메서드들을 구현하세요. 대부분의 경우 프록시는 일부 작업을 수행한 후에 그 작업을 서비스 객체에 위임해야 합니다.
4. 클라이언트가 프록시를 받을지 실제 서비스를 받을지를 결정하는 생성 메서드를 도입하는 것을 고려하세요. 이 메서드는 프록시 클래스의 간단한 정적 메서드이거나 완전한 팩토리 메서드일 수도 있습니다.
5. 서비스 객체에 대해 지연된 초기화 구현을 고려하세요.

⚠️ 장단점

- ✓ 클라이언트들이 알지 못하는 상태에서 서비스 객체를 제어할 수 있습니다.
- ✓ 클라이언트들이 신경 쓰지 않을 때 서비스 객체의 수명 주기를 관리할 수 있습니다.
- ✓ 프록시는 서비스 객체가 준비되지 않았거나 사용할 수 없는 경우에도 작동합니다.
- ✓ **개방/폐쇄 원칙**. 서비스나 클라이언트들을 변경하지 않고도 새 프록시들을 도입할 수 있습니다.
- ✗ 새로운 클래스들을 많이 도입해야 하므로 코드가 복잡해질 수 있습니다.
- ✗ 서비스의 응답이 늦어질 수 있습니다.

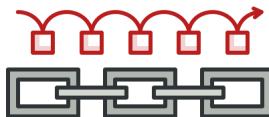
↔ 다른 패턴과의 관계

- 어댑터는 다른 인터페이스를, 프록시는 같은 인터페이스를, 데코레이터는 향상된 인터페이스를 래핑된 객체에 제공합니다.
- 퍼사드 패턴은 복잡한 객체 또는 시스템을 보호하고 자체적으로 초기화한다는 점에서 프록시와 유사합니다. 퍼사드 패턴과 달리 프록시는 자신의 서비스 객체와 같은 인터페이스를 가지므로 이들은 서로 상호 교환이 가능합니다.

- 데코레이터와 프록시의 구조는 비슷하나 이들의 의도는 매우 다릅니다. 두 패턴 모두 한 객체가 일부 작업을 다른 객체에 위임해야 하는 합성 원칙을 기반으로 합니다. 이 두 패턴의 차이점은 프록시는 일반적으로 자체적으로 자신의 서비스 객체의 수명 주기를 관리하는 반면 데코레이터의 합성은 항상 클라이언트에 의해 제어된다는 점입니다.

행동 디자인 패턴

행동 디자인 패턴들은 알고리즘들과 객체 간의 책임 할당과 관련이 있습니다.



책임 연쇄

일련의 핸들러들의 사슬을 따라 요청을 전달할 수 있게 해주는 행동 디자인 패턴입니다. 각 핸들러는 요청을 받으면 요청을 처리할지 아니면 체인의 다음 핸들러로 전달할지를 결정합니다.



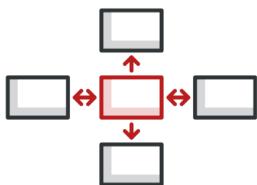
커맨드

요청을 요청에 대한 모든 정보가 포함된 독립 실행형 객체로 변환합니다. 이 변환은 다양한 요청들이 있는 메서드들을 인수화할 수 있도록 하며, 요청의 실행을 지연 또는 대기열에 넣을 수 있도록 하고, 또 실행 취소할 수 있는 작업을 지원할 수 있도록 합니다.



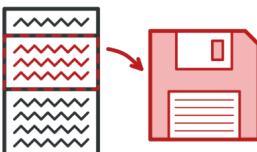
반복자

컬렉션의 요소들의 기본 표현(리스트, 스택, 트리 등)을 노출하지 않고 그들을 하나씩 순회할 수 있도록 합니다.



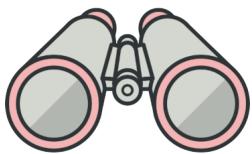
중재자

객체 간의 혼란스러운 의존 관계들을 줄일 수 있습니다. 이 패턴은 객체 간의 직접 통신을 제한하고 중재자 객체를 통해서만 협력하도록 합니다.



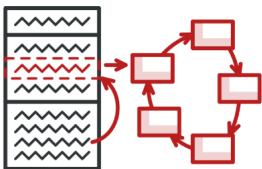
메멘토

객체의 구현 세부 사항을 공개하지 않으면서 해당 객체의 이전 상태를 저장하고 복원할 수 있게 해줍니다.



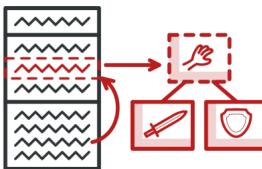
옵서버

여러 객체에 자신이 관찰 중인 객체에 발생하는 모든 이벤트에 대하여 알리는 구독 메커니즘을 정의할 수 있도록 합니다.



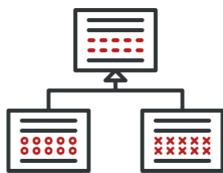
상태

객체의 내부 상태가 변경될 때 해당 객체가 그의 행동을 변경할 수 있도록 합니다. 객체가 행동을 변경할 때 객체가 클래스를 변경한 것처럼 보일 수 있습니다.



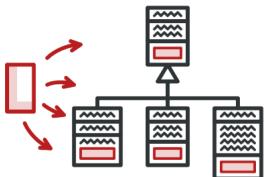
전략

알고리즘들의 패밀리를 정의하고, 각 패밀리를 별도의 클래스들에 넣은 후 그들의 객체들을 상호교환할 수 있도록 합니다.



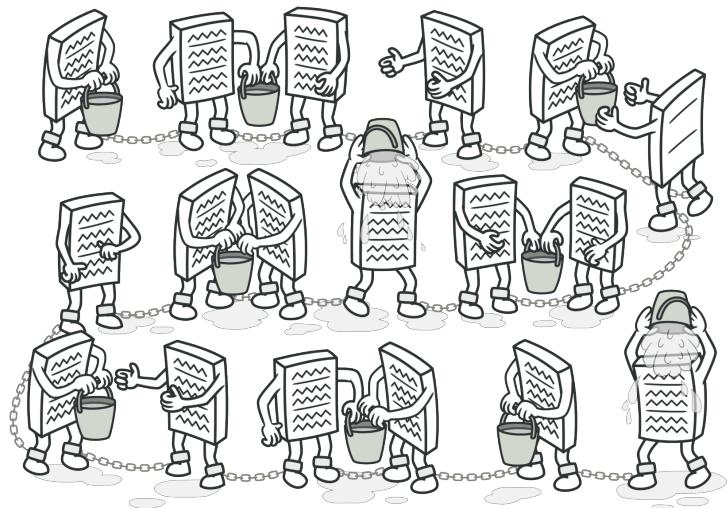
템플릿 메서드

부모 클래스에서 알고리즘의 골격을 정의하지만, 해당 알고리즘의 구조를 변경하지 않고 자식 클래스들이 알고리즘의 특정 단계들을 오버라이드(재정의)할 수 있도록 합니다.



비지터

알고리즘들을 그들이 작동하는 객체들로부터 분리할 수 있습니다.



책임 연쇄 패턴

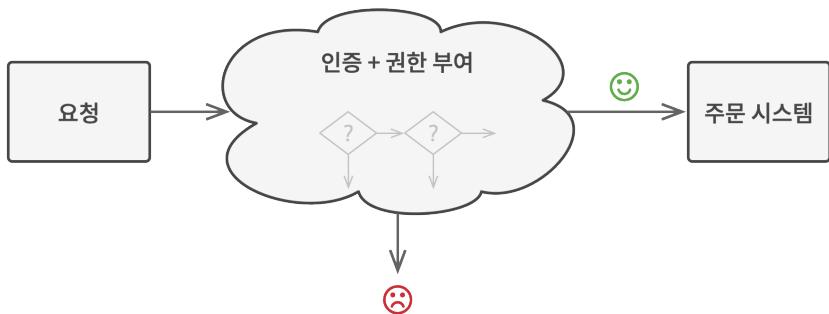
다음 이름으로도 불립니다: CoR, 커맨드 사슬, Chain of Responsibility

책임 연쇄 패턴은 핸들러들의 체인(사슬)을 따라 요청을 전달할 수 있게 해주는 행동 디자인 패턴입니다. 각 핸들러는 요청을 받으면 요청을 처리할지 아니면 체인의 다음 핸들러로 전달할지를 결정합니다.

(:(문제

당신이 온라인 주문 시스템을 개발하고 있다고 가정해봅시다. 당신은 인증된 사용자들만 주문을 생성할 수 있도록 시스템에 대한 접근을 제한하려고 합니다. 또 관리 권한이 있는 사용자들에게는 모든 주문에 대한 전체 접근 권한을 부여하려고 합니다.

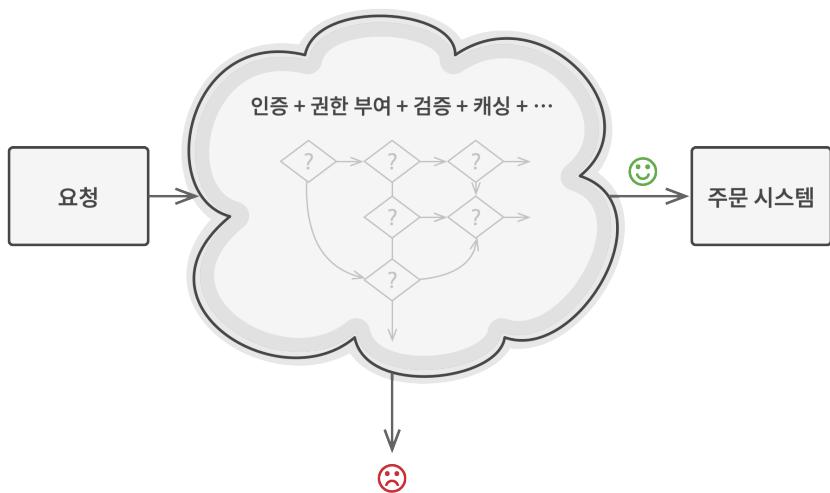
당신은 약간의 설계 후에 이러한 검사들은 차례대로 수행해야 한다는 사실을 깨달았습니다. 당신의 앱은 사용자들의 자격 증명이 포함된 요청을 받을 때마다 시스템에 대해 사용자 인증을 시도할 수 있습니다. 그러나 이러한 자격 증명이 올바르지 않아서 인증에 실패하면 다른 검사들을 진행할 이유가 없습니다.



요청은 주문 시스템 자체가 처리할 수 있기 전에 일련의 검사들을 통과해야 합니다.

다음 몇 달 동안 당신은 이러한 순차 검사들을 몇 가지 더 구현했습니다.

- 동료 중 한 명이 검증되지 않은 데이터를 주문 시스템에 직접 전달하는 것은 안전하지 않다고 제안했습니다. 그래서 당신은 요청 내의 데이터를 정제(sanitize)하는 추가 유효성 검사 단계를 추가했습니다.
- 나중에 누군가가 시스템이 무차별 대입 공격에 취약하다는 사실을 발견했으며, 이러한 공격을 방어하기 위해 같은 IP 주소에서 오는 반복적으로 실패한 요청을 걸러내는 검사를 즉시 추가했습니다.
- 또 다른 누군가는 같은 데이터가 포함된 반복 요청에 대해 캐시된 결과를 반환하여 시스템 속도를 높일 수 있다고 제안했고, 당신은 적절한 캐시 응답이 없는 경우에만 요청이 시스템으로 전달되도록 하는 또 다른 검사를 추가했습니다.



코드가 커질수록 더 복잡해졌습니다.

이미 엉망진창이었던 검사 코드는 당신이 새로운 기능을 추가할 때마다 더욱 크게 부풀어 올랐습니다. 하나의 검사 코드를 바꾸면 다른 검사 코드가 영향을 받기도 했습니다. 더 심각한 문제는, 시스템의 다른 컴포넌트들을 보호하기 위해 검사를 재사용하려고 할 때 해당 컴포넌트들에 일부 코드를 복제해야 했다는 것입니다. 왜냐하면 컴포넌트들이 필요로 한 것은 검사의 일부였지, 모든 검사는 아니었기 때문입니다.

당신의 시스템은 이해하기가 매우 어려웠고 유지 관리 비용이 많이 들었으며, 당신은 프로그램 전체를 리팩토링하기로 할 때까지 한동안 코드와 씨름했습니다.

☺ 해결책

다른 여러 행동 디자인 패턴들과 마찬가지로 **책임 연쇄** 패턴은 특정 행동들을 **핸들러**라는 독립 실행형 객체들로 변환합니다. 당신의 앱의 경우 각 검사는 검사를 수행하는 단일 메서드가 있는 자체 클래스로 추출되어야 합니다. 이제 요청은 데이터와 함께 이 메서드에 인수로 전달됩니다.

이 패턴은 이러한 핸들러들을 체인으로 연결하도록 제안합니다. 연결된 각 핸들러에는 체인의 다음 핸들러에 대한 참조를 저장하기 위한 필드가 있습니다. 요청을 처리하는 것 외에도 핸들러들은 체인을 따라 요청을 더 멀리 전달하며, 이 요청은 모든 핸들러가 요청을 처리할 기회를 가질 때까지 체인을 따라 이동합니다.

가장 좋은 부분은 핸들러가 요청을 체인 아래로 더 이상 전달하지 않고 추가 처리를 사실상 중지하는 결정을 내릴 수 있다는 것입니다.

당신의 주문 관리 시스템에서는 하나의 핸들러가 주문 처리를 수행한 다음 요청을 체인 아래로 더 전달할지를 결정합니다. 요청에 올바른 데이터가 포함되어 있다고 가정하면 모든 핸들러들은 인증 확인이든 캐싱이든 그들의 주 행동들을 실행할 수 있습니다.

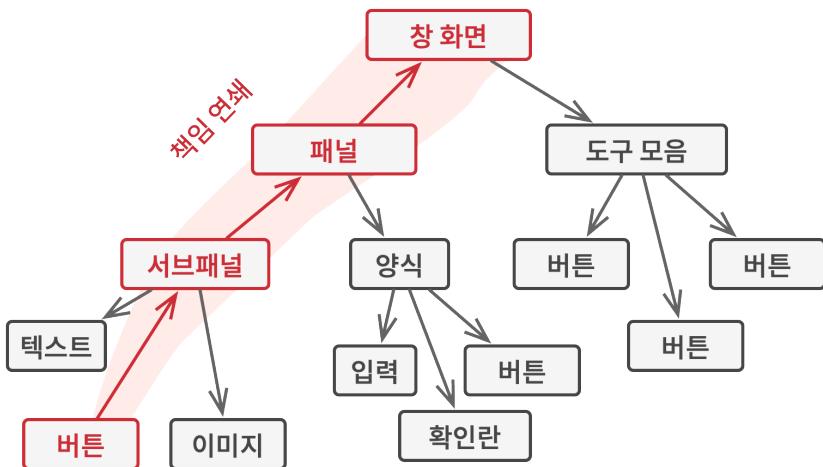


핸들러들이 하나씩 줄지어 체인을 형성합니다.

한편, 약간 다른 조금 더 정식적인 접근 방법이 있습니다. 이 방식에서는 핸들러가 요청을 받으면 핸들러는 요청을 처리할 수 있는지를 판단합니다. 처리가 가능한 경우, 핸들러는 이 요청을 더 이상 전달하지 않습니다. 따라서 요청을 처리하는 핸들러는 하나뿐이거나 아무 핸들러도 요청을 처리하지 않습니다. 이 접근 방식은 그래픽 사용자 인터페이스 내에서 요소들의 스택에서 이벤트들을 처리할 때 매우 일반적입니다.

예를 들어, 사용자가 버튼을 클릭하면 결과 이벤트는 그래픽 사용자 인터페이스 요소 체인을 통해 전파됩니다. 이 체인은

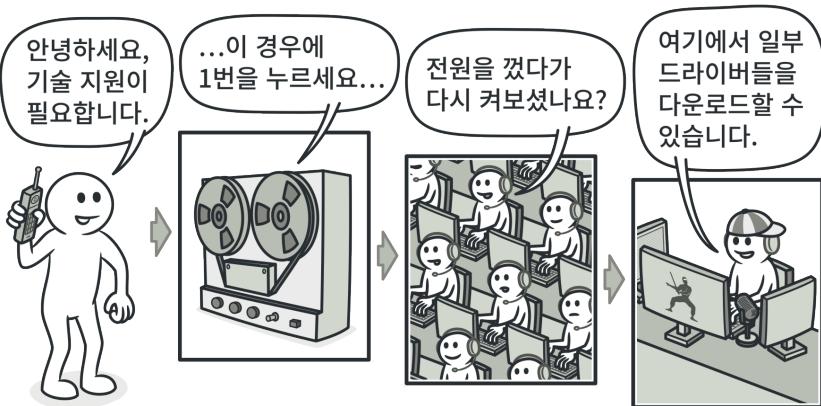
버튼으로 시작하여 해당 컨테이너들(예: 양식 또는 패널)을 따라 이동한 후 메인 애플리케이션 창으로 끝납니다. 또 이 이벤트는 그를 처리할 수 있는 체인의 첫 번째 요소에 의해 처리됩니다. 이 예가 주목할 만한 이유는 체인이 항상 객체 트리에서 추출될 수 있음을 보여주기 때문입니다.



체인은 객체 트리의 가지에서부터 형성될 수 있습니다.

모든 핸들러 클래스들이 같은 인터페이스를 구현하는 것은 매우 중요합니다. 각 구상 핸들러는 `execute` 메서드가 있는 다음 핸들러에만 신경을 써야 합니다. 이렇게 하면 다양한 핸들러들을 사용하여 코드를 핸들러들의 구상 클래스들에 결합하지 않고도 런타임에 체인들을 구성할 수 있습니다.

🚗 실제상황 적용



기술 지원 부서로의 전화는 여러 교환원을 거쳐 이루어질 수 있습니다.

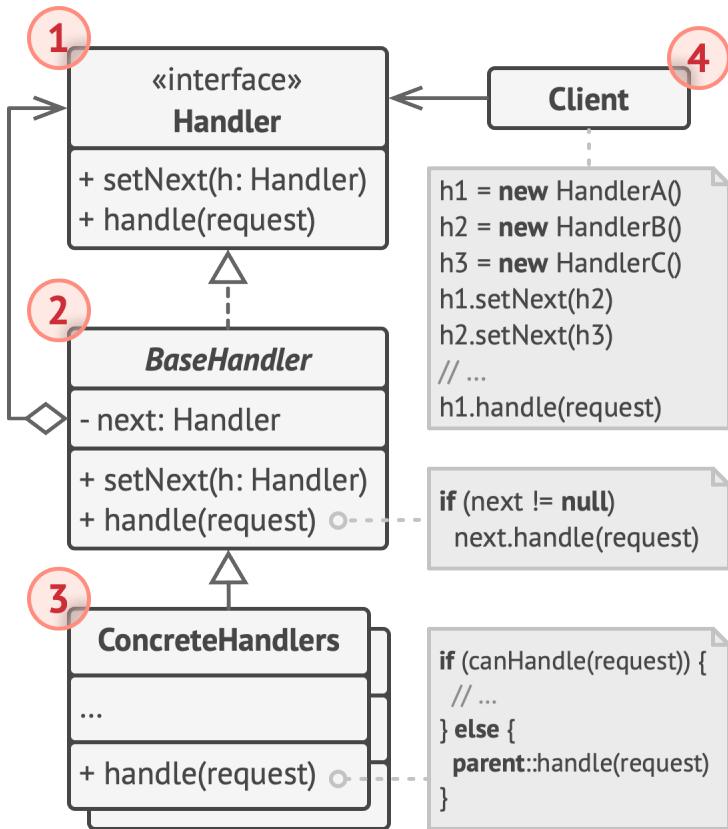
당신은 당신의 컴퓨터에 새 하드웨어를 구매하여 설치했습니다. 당신은 컴퓨터 괴짜이기 때문에 당신의 컴퓨터에는 여러 운영 체제가 설치되어 있습니다. 당신은 이 하드웨어가 지원되는지 확인하기 위해 모든 운영 체제들의 부팅을 시도합니다. 윈도우는 하드웨어를 자동으로 감지하고 활성화합니다. 그러나 당신이 애지중지하는 리눅스 운영 체제는 새 하드웨어와 작업하는 것을 거부합니다. 당신은 약간의 희망을 품고 상자에 적힌 기술 지원 전화번호로 전화하기로 합니다.

가장 먼저 들리는 것은 자동 응답기의 로봇 음성입니다. 이 음성은 다양한 문제에 대한 9가지 인기 있는 솔루션을 제안하지만, 그 중 어느 것도 당신의 문제와 관련이 없습니다. 잠시 후 로봇이 당신을 실제 교환원에게 연결합니다.

그러나 이 교환원도 별로 도움이 될만한 제안을 하지 않습니다. 그는 당신의 문제를 경청하지 않은 채 사용자 설명서에서 발췌한 긴 문장을 계속 인용합니다. '컴퓨터를 껐다가 다시 켜 보셨습니까?' 같은 별 쓸모없는 문구를 10번 이상 들은 후, 당신은 적절한 엔지니어와 연결해 줄 것을 요구합니다.

결국 드디어 교환원은 어둡고 외로운 지하 서버실에서 인간적인 접촉을 갈망했던 엔지니어 중 한 명에게 전화를 연결합니다. 이 엔지니어는 새 하드웨어에 적합한 드라이브를 어디에서 다운받아야 하는지와 리눅스에 설치하는 방법 등을 알려줍니다. 드디어 문제가 해결되었군요! 당신은 기쁜 마음으로 통화를 종료합니다.

구조



1. **핸들러는** 모든 구상 핸들러에 공통적인 인터페이스를 선언합니다. 일반적으로 여기에는 요청을 처리하기 위한 단일 메서드만 포함되지만 때로는 체인의 다음 핸들러를 세팅하기 위한 다른 메서드가 있을 수도 있습니다.
2. **기초 핸들러는** 선택적 클래스이며 여기에 모든 핸들러 클래스들에 공통적인 상용구 코드를 넣을 수 있습니다.

일반적으로 이 클래스는 다음 핸들러에 대한 참조를 저장하기 위한 필드를 정의합니다. 클라이언트들은 핸들러를 이전 핸들러의 생성자 또는 세터(setter)에 해당 핸들러를 전달하여 체인을 구축할 수 있습니다. 또 클래스는 디폴트 핸들러 행동을 구현할 수도 있습니다. 즉, 다음 핸들러의 존재 여부를 확인한 후 다음 핸들러로 실행을 넘길 수 있습니다.

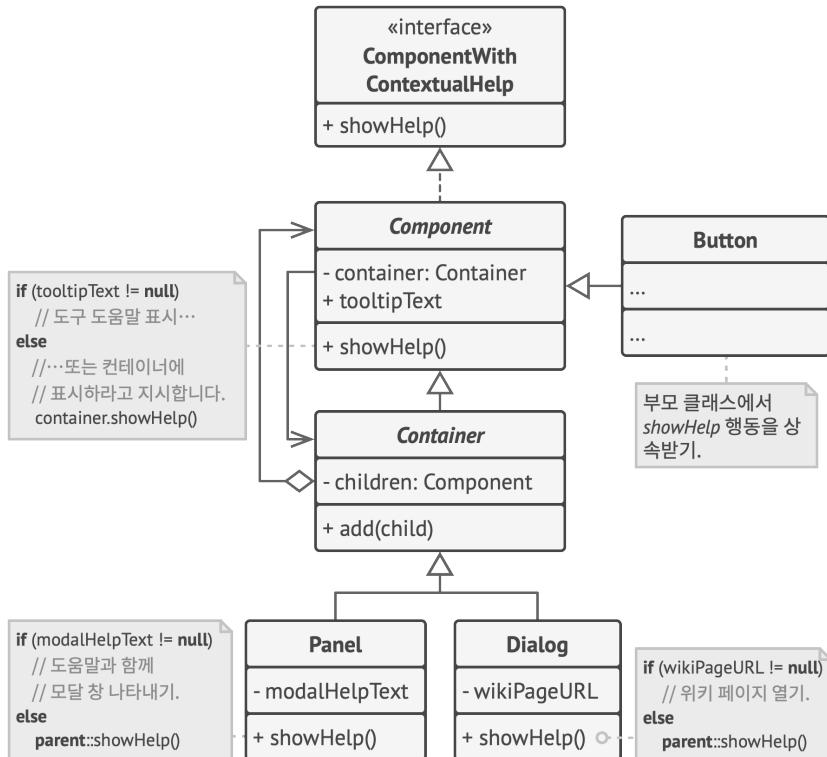
3. **구상 핸들러들**에는 요청을 처리하기 위한 실제 코드가 포함되어 있습니다. 각 핸들러는 요청을 받으면 이 요청을 처리할지와 함께 체인을 따라 전달할지를 결정해야 합니다.

핸들러들은 일반적으로 자체 포함형이고 불변하며, 생성자를 통해 필요한 모든 데이터를 한 번만 받습니다.

4. **클라이언트**는 앱의 논리에 따라 체인들을 한 번만 구성하거나 동적으로 구성할 수 있습니다. 참고로 요청은 체인의 모든 핸들러에 보낼 수 있으며, 꼭 첫 번째 핸들러일 필요는 없습니다.

의사코드

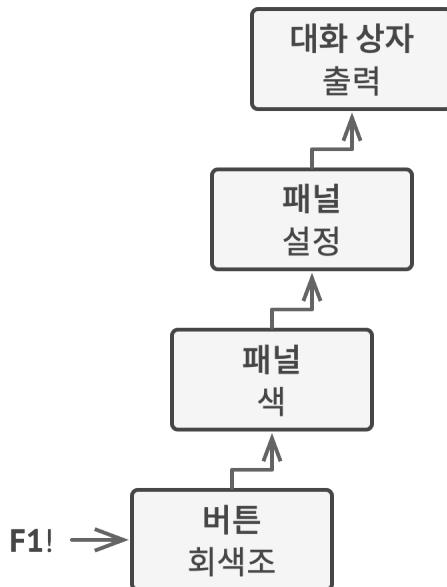
이 예에서 **책임 연쇄** 패턴은 활성 그래픽 사용자 인터페이스 요소에 대한 상황별 도움말 정보를 표시하는 역할을 합니다.



그래픽 사용자 인터페이스 클래스들은 복합체 패턴으로 빌드되고, 각 요소는 그 요소의 컨테이너 요소에 연결됩니다. 또 언제든지 요소 자체에서 시작하여 그 요소의 모든 컨테이너 요소를 통과하는 요소 체인을 구축할 수 있습니다.

앱의 그래픽 사용자 인터페이스의 구조는 일반적으로 객체 트리로 구성됩니다. 예를 들어 앱의 기본 창을 렌더링하는 `Dialog` (대화 상자) 클래스는 객체 트리의 뿌리 (root)가 됩니다. `Dialog`에는 `Panels` (패널들)가 포함되어 있으며, 여기에는 다른 패널들이나 `Buttons` (버튼들) 및 `TextFields` (문자 필드들)와 같은 단순한 하위 설계 요소들이 포함될 수 있습니다.

간단한 컴포넌트는 컴포넌트에 어떤 도움말 텍스트가 할당되어 있는 한 상황에 맞는 짧은 도구 도움말들을 표시할 수 있습니다. 그러나 더 복잡한 컴포넌트들은 (예를 들어 설명서에서 발췌한 내용을 표시하거나 브라우저에서 웹페이지를 여는 것과 같은 상황에 맞는 도움말을 표시하는 컴포넌트들) 상황별 도움말을 나타내기 위한 그들의 고유한 방법들을 정의할 수 있습니다.



도움말 요청이 그래픽 사용자 인터페이스 객체들을 가로질러 이동하는 방법입니다.

사용자가 요소에 마우스 커서를 놓고 F1 키를 누르면 앱은 포인터 아래에 있는 컴포넌트를 감지하고 그에게 도움 요청을 보냅니다. 이 요청은 도움말 정보를 표시할 수 있는 요소에 도달할 때까지 모든 요소의 컨테이너를 통과하며 올라갑니다.

```

1 // 핸들러 인터페이스는 요청을 실행하기 위한 메서드를 선언합니다.
2 interface ComponentWithContextualHelp is
3     method showHelp()
4
5
6 // 간단한 컴포넌트들의 기초 클래스.
7 abstract class Component implements ComponentWithContextualHelp is
8     field tooltipText: string
9
10 // 컴포넌트의 컨테이너는 핸들러 체인의 다음 링크 역할을 합니다.
11 protected field container: Container
12
13 // 컴포넌트는 도움말 텍스트가 할당되었을 때 도구 설명을 표시합니다. 그렇지
14 // 않으면 컨테이너가 있는 경우 호출을 해당 컨테이너로 전달합니다.
15 method showHelp() is
16     if (tooltipText != null)
17         // 도구 설명 표시하기.
18     else
19         container.showHelp()
20
21
22 // 컨테이너는 간단한 컴포넌트들과 다른 컨테이너들을 자식으로 포함할 수 있습니다.
23 // 여기에서 체인 관계들이 설립됩니다. 이 클래스는 부모로부터 showHelp 행동을
24 // 상속합니다.
25 abstract class Container extends Component is
26     protected field children: array of Component
27
28 method add(child) is
29     children.add(child)
30     child.container = this
31
32

```

```
33 // 원시적인 컴포넌트들은 디폴트 도움말 구현으로 괜찮을 수 있습니다...
34 class Button extends Component is
35     // ...
36
37 // 그러나 복잡한 컴포넌트들은 기초 구현을 오버라이드할 수 있습니다. 도움말 텍스트를
38 // 새로운 방식으로 제공할 수 없는 경우 컴포넌트는 언제든지 기초 구현을 호출할 수
39 // 있습니다. (컴포넌트 클래스 참조).
40 class Panel extends Container is
41     field modalHelpText: string
42
43     method showHelp() is
44         if (modalHelpText != null)
45             // 도움말 텍스트와 함께 모달 창을 표시합니다.
46         else
47             super.showHelp()
48
49 // ...위와 같음...
50 class Dialog extends Container is
51     field wikiPageURL: string
52
53     method showHelp() is
54         if (wikiPageURL != null)
55             // 위키 도움말 페이지를 엽니다.
56         else
57             super.showHelp()
58
59
60 // 클라이언트 코드
61 class Application is
62     // 모든 앱은 체인을 다르게 설정합니다.
63     method createUI() is
64         dialog = new Dialog("Budget Reports")
```

```

65     dialog.wikiPageURL = "http://..."
66     panel = new Panel(0, 0, 400, 800)
67     panel.modalHelpText = "This panel does..."
68     ok = new Button(250, 760, 50, 20, "OK")
69     ok.tooltipText = "This is an OK button that..."
70     cancel = new Button(320, 760, 50, 20, "Cancel")
71     // ...
72     panel.add(ok)
73     panel.add(cancel)
74     dialog.add(panel)
75
76 // 여기에서 무슨 일이 일어날지 상상해 보세요.
77 method onF1KeyPress() is
78     component = this.getComponentAtMouseCoords()
79     component.showHelp()

```

💡 적용

 책임 연쇄 패턴은 당신의 프로그램이 다양한 방식으로 다양한 종류의 요청들을 처리할 것으로 예상되지만 정확한 요청 유형들과 순서들을 미리 알 수 없는 경우에 사용하세요.

 이 패턴은 당신이 여러 핸들러를 하나의 체인으로 연결할 수 있도록 해주고, 또 요청을 받으면 바로 각 핸들러에게 이 요청을 처리할 수 있는지 질문합니다. 이렇게 해야 모든 핸들러들은 요청을 처리할 기회를 얻습니다.

 이 패턴은 특정 순서로 여러 핸들러를 실행해야 할 때 사용하세요.

- ↳ 당신은 체인의 핸들러들을 원하는 순서로 연결할 수 있으므로 모든 요청은 정확히 당신이 계획한 대로 체인을 통과합니다.

- ↳ 책임 연쇄 패턴은 핸들러들의 집합과 그들의 순서가 런타임에 변경되어야 할 때 사용하세요.

- ↳ 당신이 핸들러 클래스들 내부의 참조 필드에 세터들을 제공하면, 핸들러들을 동적으로 삽입, 제거 또는 재정렬할 수 있을 것입니다.

▣ 구현방법

1. 핸들러 인터페이스를 선언하고 요청을 처리하는 메서드의 시그니처를 설명하세요.

클라이언트가 요청 데이터를 메서드에 전달하는 방법을 결정하세요. 가장 유연한 방법은 요청을 객체로 변환하여 처리 메서드에 인수로 전달하는 것입니다.

2. 구상 핸들러들에서 중복된 상용구 코드를 제거하려면 핸들러 인터페이스에서 파생된 추상 기초 핸들러 클래스를 만드는 것도 고려해볼 만합니다.

이 클래스에는 체인의 다음 핸들러에 대한 참조를 저장하기 위한 필드가 있어야 합니다. 이 클래스를 불변으로 만드는 것을

고려하세요. 그러나 런타임에 체인들을 수정할 계획이라면 참조 필드의 값을 변경하기 위한 세터(setter)를 정의해야 합니다.

당신은 또 처리(핸들링) 메서드를 위한 편리한 디플트(기본값) 행동을 구현할 수 있으며, 이 행동은 남아있는 객체가 없을 때까지 요청을 다음 객체로 넘기는 것입니다. 구상 핸들러들은 부모 메서드를 호출하여 이 행동을 사용할 수 있습니다.

3. 하나씩 구상 핸들러 자식 클래스들을 만들고 그들의 처리 메서드들을 구현하세요. 각 핸들러는 요청을 받았을 때 두 가지 결정을 내려야 합니다:
 - 요청을 처리할지의 여부.
 - 체인을 따라 요청을 전달할지의 여부.
4. 클라이언트는 자체적으로 체인을 조립하거나 다른 객체들에서부터 미리 구축된 체인을 받을 수 있습니다. 후자의 경우 설정 또는 환경 설정에 따라 체인들을 구축하기 위해 일부 공장 클래스들을 구현해야 합니다.
5. 클라이언트는 첫 번째 핸들러뿐만 아니라 체인의 모든 핸들러를 활성화할 수 있습니다. 요청은 어떤 핸들러가 더 이상의 전달을 거부하거나 요청이 체인 끝에 도달할 때까지 체인을 따라 전달됩니다.

6. 체인의 동적 특성으로 인해 클라이언트는 다음 상황들을 처리할 준비가 되어 있어야 합니다:

- 체인은 단일 링크로 구성될 수 있습니다.
- 일부 요청들은 체인 끝에 도달하지 못할 수 있습니다.
- 다른 요청들은 처리되지 않은 상태로 체인의 끝에 도달할 수 있습니다.

⚠ 장단점

- ✓ 요청의 처리 순서를 제어할 수 있습니다.
- ✓ 단일 책임 원칙. 당신은 작업을 호출하는 클래스들을 작업을 수행하는 클래스들과 분리할 수 있습니다.
- ✓ 개방/폐쇄 원칙. 기존 클라이언트 코드를 손상하지 않고 앱에 새 핸들러들을 도입할 수 있습니다.
- ✗ 일부 요청들은 처리되지 않을 수 있습니다.

↔ 다른 패턴과의 관계

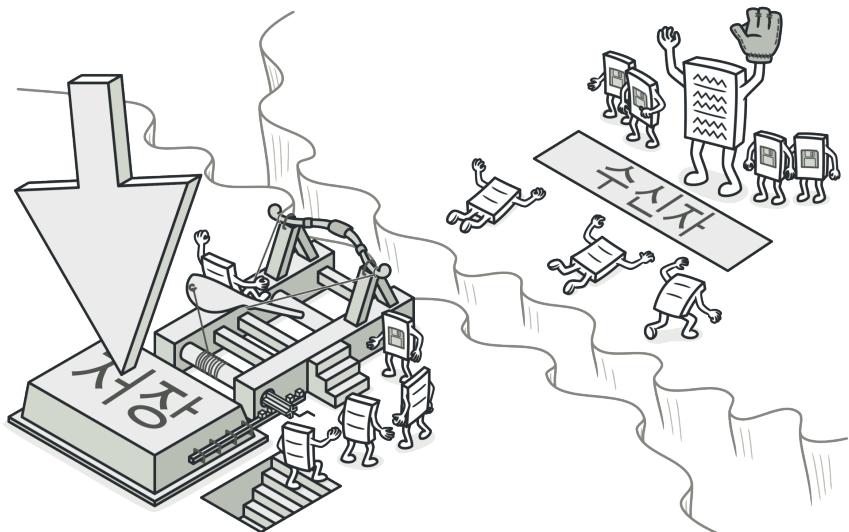
- 커맨드, 중재자, 옵서버 및 책임 연쇄 패턴은 요청의 발신자와 수신자를 연결하는 다양한 방법을 다룹니다.

- 책임 연쇄 패턴은 잠재적 수신자의 동적 체인을 따라 수신자 중 하나에 의해 요청이 처리될 때까지 요청을 순차적으로 전달합니다.
 - 커맨드 패턴은 발신자와 수신자 간의 단방향 연결을 설립합니다.
 - 중재자 패턴은 발신자와 수신자 간의 직접 연결을 제거하여 그들이 중재자 객체를 통해 간접적으로 통신하도록 강제합니다.
 - 읍서버 패턴은 수신자들이 요청들의 수신을 동적으로 구독 및 구독 취소할 수 있도록 합니다.
- 책임 연쇄 패턴은 종종 복합체 패턴과 함께 사용됩니다. 그러면 임 컴포넌트가 요청을 받으면 해당 요청을 모든 부모 컴포넌트들의 체인을 통해 객체 트리의 뿌리(root)까지 전달할 수 있습니다.
 - 책임 연쇄 패턴의 핸들러들은 커맨드로 구현할 수 있습니다. 그러면 당신은 많은 다양한 작업을 같은 콘텍스트 객체에 대해 실행할 수 있으며, 해당 콘텍스트 객체는 요청의 역할을 합니다. 여기에서의 요청은 처리 메서드의 매개변수를 의미합니다.

그러나 요청 자체가 커맨드 객체인 다른 접근 방식이 있습니다. 이 접근 방식을 사용하면 당신은 같은 작업을 체인에 연결된 일련의 서로 다른 콘텍스트들에서 실행할 수 있습니다.

- **책임 연쇄** 패턴과 **데코레이터**는 클래스 구조가 매우 유사합니다. 두 패턴 모두 실행을 일련의 객체들을 통해 전달할 때 재귀적인 합성에 의존하나, 몇 가지 결정적인 차이점이 있습니다.

책임 연쇄 패턴 핸들러들은 서로 독립적으로 임의의 작업을 실행할 수 있으며, 또한 해당 요청을 언제든지 더 이상 전달하지 않을 수 있습니다. 반면에 다양한 **데코레이터**들은 객체의 행동을 확장하며 동시에 이러한 행동을 기초 인터페이스와 일관되게 유지할 수 있습니다. 또한 **데코레이터**들은 요청의 흐름을 중단할 수 없습니다.



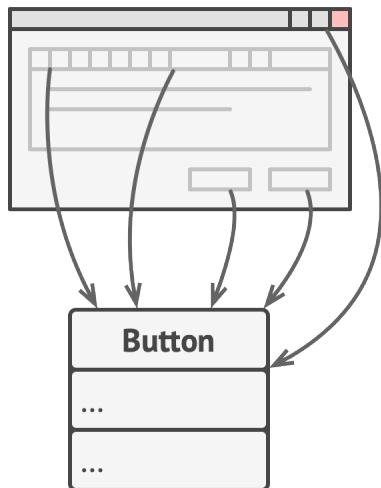
커맨드 패턴

다음 이름으로도 불립니다: 액션, 트랜잭션, Command

커맨드는 요청을 요청에 대한 모든 정보가 포함된 독립실행형 객체로 변환하는 행동 디자인 패턴입니다. 이 변환은 다양한 요청들이 있는 메서드들을 인수화 할 수 있도록 하며, 요청의 실행을 지연 또는 대기열에 넣을 수 있도록 하고, 또 실행 취소할 수 있는 작업을 지원할 수 있도록 합니다.

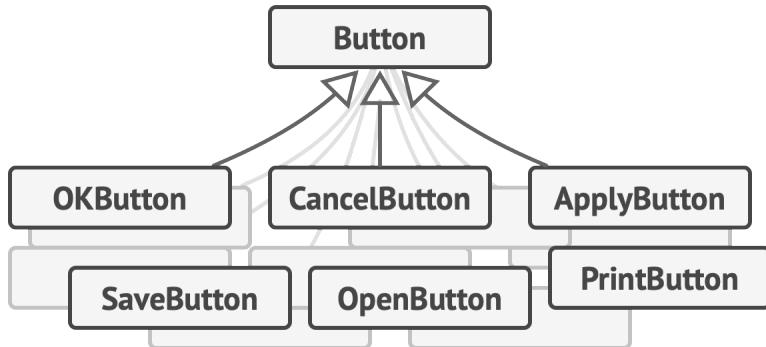
(:) 문제

당신이 새로운 텍스트 편집기 앱을 개발하고 있다고 상상해 봅시다. 당신이 현재 하는 작업은 편집기의 다양한 작업을 위한 여러 버튼이 있는 도구 모음(툴바)을 만드는 것입니다. 당신은 도구 모음의 버튼들과 다양한 대화 상자들의 일반 버튼들에 사용할 수 있는 매우 깔끔한 `Button` (버튼) 클래스를 만들었습니다.



앱의 모든 버튼은 같은 클래스에서 파생됩니다.

이 버튼들은 모두 비슷해 보이지만 각각 다른 기능들을 수행해야 합니다. 그러면 이 버튼들의 다양한 클릭 핸들러들에 대한 코드는 어디에 두겠습니까? 가장 간단한 해결책은 버튼이 사용되는 각 위치에 수많은 자식 클래스들을 만드는 것입니다. 이러한 자식 클래스들에는 버튼 클릭 시 실행되어야 하는 코드가 포함됩니다.



많은 버튼 자식 클래스들이 있습니다. 무엇이 잘못될 수 있을까요?

머지않아 당신은 이 접근 방식에 심각한 결함이 있음을 깨닫게 됩니다. 일단, 당신은 이제 엄청난 수의 자식 클래스들이 있으며, 기초 `Button` 클래스를 수정할 때마다 이러한 자식 클래스의 코드를 깨뜨릴 위험이 있습니다. 간단히 말해서, 그래픽 사용자 인터페이스 코드는 비즈니스 로직의 불안정한 코드에 어색하게 의존하게 되었습니다.



여러 클래스가 같은 기능을 구현합니다.

그리고 당신이 고려해야 할 최악의 사항은, 텍스트 복사/붙여넣기와 같은 일부 작업은 여러 위치에서 호출될 수 있다는 사실입니다. 예를 들어, 사용자는 무언가를 복사하기 위하여 도구

모음에서 작은 '복사' 버튼을 클릭하거나 콘텍스트 메뉴를 통해 무언가를 복사하거나 키보드에서 `Ctrl+C` 를 누를 수 있습니다.

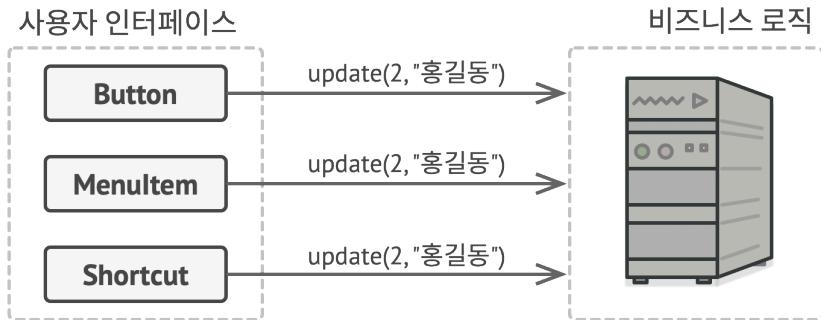
당신의 앱에 처음에 하나의 도구 모음만 있었을 때는 다양한 작업의 구현을 버튼의 자식 클래스들에 배치해도 괜찮았습니다. 즉, `CopyButton` 자식 클래스의 내부에 텍스트를 복사하는 코드가 있어도 괜찮았습니다. 그러나 복사를 할 수 있도록 하는 콘텍스트 메뉴, 바로 가기 및 기타 항목들을 구현하면 당신은 이제 해당 작업의 코드를 많은 클래스에 복제하거나 버튼에 의존하는 메뉴들을 만들어야 하는데, 이것은 오히려 더 나쁜 옵션입니다.

都有自己解决问题

올바른 소프트웨어 디자인은 종종 관심사 분리의 원칙을 기반으로 합니다. 가장 일반적인 예로는 그래픽 사용자 인터페이스용 레이어와 비즈니스 로직용 레이어의 분리입니다. 그래픽 사용자 인터페이스용 레이어는 모든 입력을 캡처하고 화면에 아름다운 그림을 렌더링하고 사용자와 앱이 수행하는 작업의 결과를 나타내는 역할을 합니다. 그러나 달의 행성 궤도를 계산하거나 연간 보고서를 작성하는 것과 같은 중요한 작업을 수행할 때 그래픽 사용자 인터페이스 레이어는 비즈니스 논리의 배경 레이어들에 작업을 위임합니다.

위의 내용은 코드로 다음과 같이 표현될 수 있습니다. 그래픽 사용자 인터페이스 객체가 비즈니스 논리 객체의 메서드를

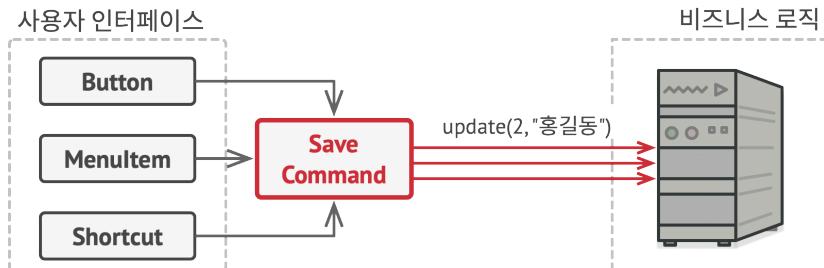
호출하고 일부 인수를 전달합니다. 위 프로세스는 일반적으로 한 객체가 다른 객체에 요청을 보내는 것이라고 불립니다.



그래픽 사용자 인터페이스 객체들은 비즈니스 논리 객체들에 직접 접근할 수 있습니다.

커맨드 패턴은 그래픽 사용자 인터페이스 객체들이 이러한 요청을 직접 보내서는 안된다고 합니다. 대신 모든 요청 세부 정보들(예: 호출되는 객체, 메서드 이름 및 인수 리스트)을 요청을 작동시키는 단일 메서드를 가진 별도의 커맨드 클래스로 추출하라고 제안합니다.

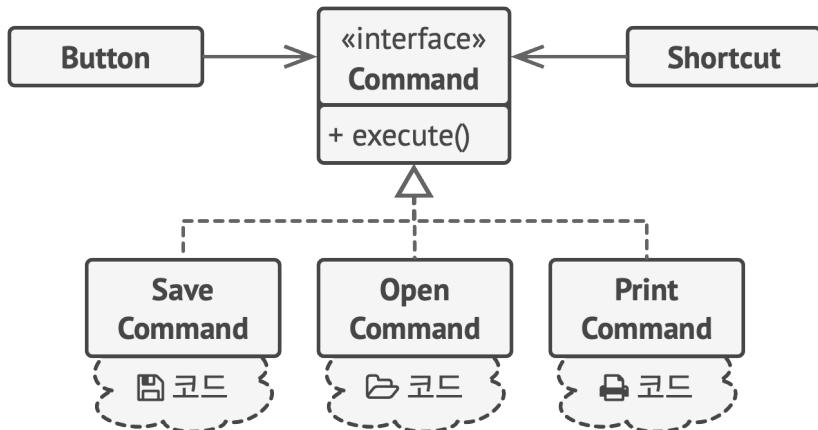
커맨드 객체들은 다양한 그래픽 사용자 인터페이스 객체들과 비즈니스 논리 객체들 간의 링크 역할을 합니다. 이제부터 그래픽 사용자 인터페이스 객체는 어떤 비즈니스 논리 객체가 요청을 받을지와 이 요청이 어떻게 처리할지에 대하여 알 필요가 없습니다. 그래픽 사용자 인터페이스 객체는 커맨드를 작동시킬 뿐이며, 그렇게 작동된 커맨드는 모든 세부 사항을 처리합니다.



커맨드를 통해 비즈니스 논리 레이어를 접근합니다.

이제 다음 단계는 당신의 커맨드들이 같은 인터페이스를 구현하도록 하는 것입니다. 일반적으로 커맨드는 매개 변수들을 받지 않는 단일 실행 메서드만을 가집니다. 이 인터페이스는 다양한 커맨드들을 커맨드들의 구상 클래스들과 결합하지 않고 같은 요청 발신자와 사용할 수 있게 해줍니다. 이제 당신은 발신자에 연결된 커맨드 객체들을 전환할 수 있으며, 그렇게 하여 런타임에 발신자의 행동을 변경할 수 있습니다.

당신은 요청 매개변수들이 빠져있다는 점을 눈치채셨을 것입니다. 그래픽 사용자 인터페이스 객체가 비즈니스 레이어 객체에 일부 매개변수들을 제공했을 수 있습니다. 커맨드 실행 메서드에 매개변수들이 없는데, 그러면 어떻게 요청의 세부 정보를 수신자에게 전달할 수 있을까요? 커맨드를 이러한 데이터로 미리 설정해놓거나, 이 데이터를 자체적으로 가져올 수 있도록 해야 합니다.



그래픽 사용자 인터페이스 객체들은 작업을 커맨드들에 위임합니다.

다시 당신의 텍스트 편집기를 살펴봅시다. 커맨드 패턴을 적용한 후에는 더 이상 다양한 클릭 행동들을 구현하기 위한 여러 버튼 자식 클래스들이 필요하지 않습니다. 기초 **Button** 클래스에 커맨드 객체에 대한 참조를 저장하는 단일 필드를 넣은 후 이 버튼이 클릭 될 때 그 커맨드를 시행하도록 하면 됩니다.

이제 가능한 모든 작업에 대해 많은 커맨드 클래스들을 구현하고 이 클래스들을 버튼의 의도된 동작에 따라 특정 버튼들과 연결해야 합니다.

메뉴, 단축키 또는 대화 상자와 같은 다른 그래픽 사용자 인터페이스 요소들도 같은 방식으로 구현할 수 있습니다. 이들은 사용자가 그래픽 사용자 인터페이스 요소와 상호 작용할 때 실행되는 커맨드에 연결될 것입니다. 지금쯤 짐작하셨겠지만 같은

작업과 관련된 요소들은 같은 커맨드들에 연결되어 코드 중복을 방지할 것입니다.

결과적으로 커맨드들은 그래픽 사용자 인터페이스 레이어와 비즈니스 로직 레이어 간의 결합도를 줄이는 편리한 중간 레이어들이 됩니다. 그리고 이것은 커맨드 패턴이 제공할 수 있는 이점의 극히 일부에 불과합니다!

🚗 실제상황 적용



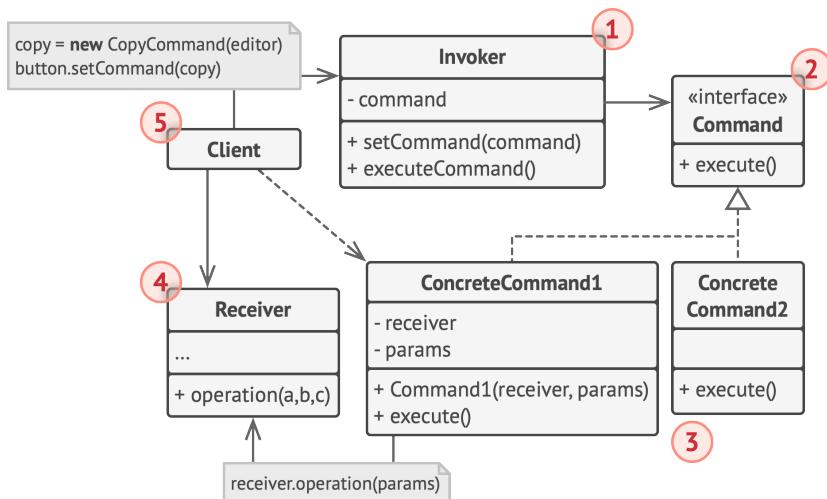
레스토랑에서 주문하기.

당신은 도시를 한참 걷다가 멋진 레스토랑에 도착하여 창가 테이블에 앉습니다. 친절한 웨이터가 다가와 신속하게 당신의 주문을 받아 종이에 적습니다. 웨이터는 부엌으로 가서 주문을 벽에 붙입니다. 잠시 후 요리사에게 주문이 전달되고 요리사는 주문을 읽고 그에 따라 음식을 요리합니다. 요리사는 주문과 함께 식사를 트레이에 놓습니다. 웨이터는 트레이를 발견한 후 당신이

주문한 대로 식사가 요리되었는지 확인하고 완성된 주문을 당신의 테이블로 가져옵니다.

종이에 적힌 주문은 커맨드 역할을 합니다. 이 주문은 요리사가 요리할 준비가 될 때까지 대기열에 남아 있습니다. 주문에는 식사를 요리하는 데 필요한 모든 관련 정보가 포함되어 있습니다. 이를 통해 요리사는 당신에게서 주문 세부 사항을 직접 전달받는 대신 바로 요리를 시작할 수 있습니다.

구조



1. **발송자** 클래스 (*invoker*라고도 함)는 요청들을 시작하는 역할을 합니다. 이 클래스에는 커맨드 객체에 대한 참조를 저장하기 위한 필드가 있어야 합니다. 발송자는 요청을 수신자에게 직접 보내는 대신 해당 커맨드를 작동시킵니다. 참고로 발송자는 커맨드

객체를 생성할 책임이 없으며 일반적으로 생성자를 통해 클라이언트로부터 미리 생성된 커맨드를 받습니다.

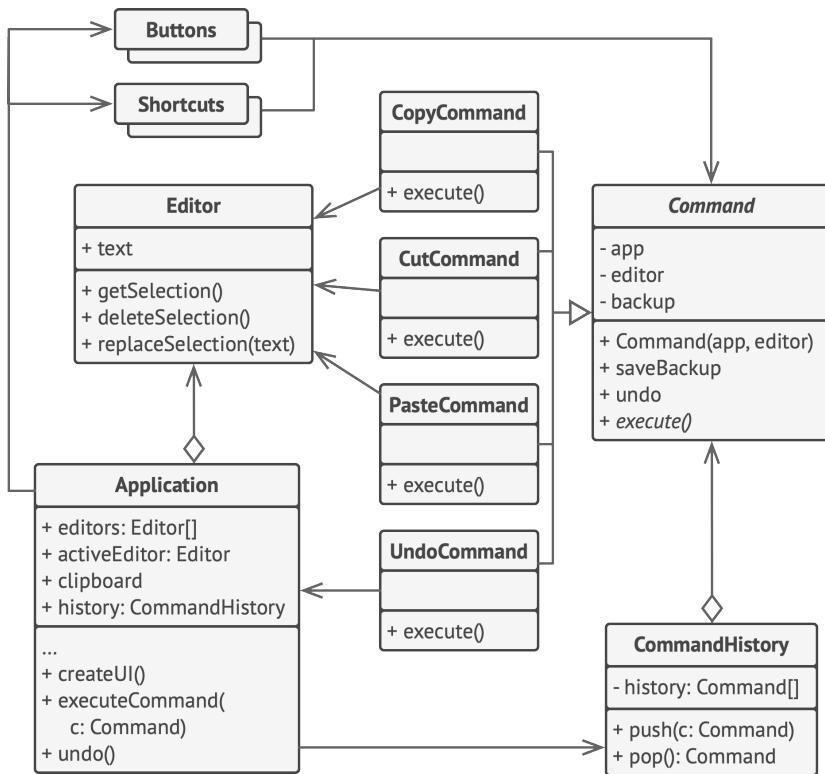
2. **커맨드** 인터페이스는 일반적으로 커맨드를 실행하기 위한 단일 메서드만을 선언합니다.
3. **구상 커맨드**들은 다양한 유형의 요청을 구현합니다. 구상 커맨드는 자체적으로 작업을 수행해서는 안 되며, 대신 비즈니스 논리 객체 중 하나에 호출을 전달해야 합니다. 그러나 코드를 단순화하기 위해 이러한 클래스들은 병합될 수 있습니다.

수신 객체에서 메서드를 실행하는 데 필요한 매개 변수들은 구상 커맨드의 필드들로 선언할 수 있습니다. 생성자를 통해서만 이러한 필드들의 초기화를 허용함으로써 커맨드 객체들을 불변으로 만들 수 있습니다.

4. **수신자** 클래스에는 일부 비즈니스 로직이 포함되어 있습니다. 거의 모든 객체는 수신자 역할을 할 수 있습니다. 대부분의 커맨드들은 요청이 수신자에게 전달되는 방법에 대한 세부 정보만 처리하는 반면 수신자 자체는 실제 작업을 수행합니다.
5. **클라이언트**는 구상 커맨드 객체들을 만들고 설정합니다. 클라이언트는 수신자 인스턴스를 포함한 모든 요청 매개변수들을 커맨드의 생성자로 전달해야 하며 그렇게 만들어진 커맨드는 하나 또는 여러 발송자와 연관될 수 있습니다.

의사코드

이 예시에서 **커맨드** 패턴은 실행된 작업의 기록을 추적하는 데 도움을 주며 필요한 경우 작업을 되돌릴 수 있도록 합니다.



텍스트 편집기에서 실행 취소될 수 있는 작업들.

잘라내기 및 붙여넣기 등의 편집기 상태를 변경하는 커맨드들은 커맨드와 관련된 작업을 실행하기 전에 편집기 상태의 백업 복사본을 만듭니다. 어떤 커맨드가 실행되고 나면, 그 커맨드는 해당 지점에서 편집기 상태의 백업 복사본과 함께 커맨드 기록

(커맨드 객체들의 스택)에 배치됩니다. 나중에 사용자가 작업을 되돌려야 하는 경우 앱은 기록에서 가장 최근 커맨드를 가져와 편집기 상태의 관련 백업을 읽은 후 작업을 되돌릴 수 있습니다.

클라이언트 코드(그래픽 사용자 인터페이스 요소들, 커맨드 기록 등)는 커맨드 인터페이스를 통해 커맨드와 함께 작동하기 때문에 구상 커맨드 클래스들에 결합하지 않습니다. 이러한 접근 방식을 사용하면 기존 코드를 손상하지 않고 앱에 새 커맨드들을 도입할 수 있습니다.

```

1 // 기초 커맨드 클래스는 모든 구상 커맨드에 대한 공통 인터페이스를 정의합니다.
2 abstract class Command is
3     protected field app: Application
4     protected field editor: Editor
5     protected field backup: text
6
7     constructor Command(app: Application, editor: Editor) is
8         this.app = app
9         this.editor = editor
10
11     // 편집기의 상태에 대한 백업을 만드세요.
12     method saveBackup() is
13         backup = editor.text
14
15     // 편집기의 상태를 복원하세요.
16     method undo() is
17         editor.text = backup
18
19     // 실행 메서드는 모든 구상 커맨드들이 자체 구현을 제공하도록 강제하기 위해

```

```
20 // 추상으로 선언됩니다. 이 메서드는 커맨드가 편집기의 상태를 변경하는지에 따라
21 // 진실 또는 거짓을 반환해야 합니다.
22 abstract method execute()
23
24
25 // 구상 커맨드들은 여기에 배치됩니다.
26 class CopyCommand extends Command is
27     // 복사 커맨드는 편집기의 상태를 변경하지 않으므로 기록에 저장되지 않습니다.
28     method execute() is
29         app.clipboard = editor.getSelection()
30         return false
31
32 class CutCommand extends Command is
33     // cut 커맨드는 편집기의 상태를 변경하므로 기록에 반드시 저장되어야 하며,
34     // 메서드가 true를 반환하는 한 저장됩니다.
35     method execute() is
36         saveBackup()
37         app.clipboard = editor.getSelection()
38         editor.deleteSelection()
39         return true
40
41 class PasteCommand extends Command is
42     method execute() is
43         saveBackup()
44         editor.replaceSelection(app.clipboard)
45         return true
46
47 // 실행취소 작업도 커맨드입니다.
48 class UndoCommand extends Command is
49     method execute() is
50         app.undo()
51         return false
```

```
52
53
54 // 글로벌 커맨드 기록도 스택일 뿐입니다.
55 class CommandHistory is
56     private field history: array of Command
57
58     // 후입 ...
59     method push(c: Command) is
60         // 커맨드를 기록 배열의 끝으로 푸시하세요.
61
62     // ... 선출
63     method pop():Command is
64         // 기록에서 가장 최근 명령을 가져오세요.
65
66
67 // 편집기 클래스에는 실제 텍스트 편집 기능이 있습니다. 이는 수신기의 역할을
68 // 합니다. 모든 커맨드들은 결국 편집기의 메서드들에 실행을 위임합니다.
69 class Editor is
70     field text: string
71
72     method getSelection() is
73         // 선택된 텍스트를 반환하세요.
74
75     method deleteSelection() is
76         // 선택된 텍스트를 삭제하세요.
77
78     method replaceSelection(text) is
79         // 현재 위치에 클립보드의 내용을 삽입하세요.
80
81
82 // 앱 클래스는 객체 관계들을 설정하며 발신자 역할을 합니다. 이것은 무언가를
83 // 수행해야 할 때 커맨드 객체를 만들고 실행합니다.
```

```

84  class Application is
85      field clipboard: string
86      field editors: array of Editors
87      field activeEditor: Editor
88      field history: CommandHistory
89
90      // 사용자 인터페이스 객체들에 커맨드들을 할당하는 코드는 다음과 같을 수
91      // 있습니다.
92      method createUI() is
93          // ...
94          copy = function() { executeCommand(
95              new CopyCommand(this, activeEditor)) }
96          copyButton.setCommand(copy)
97          shortcuts.onKeyPress("Ctrl+C", copy)
98
99          cut = function() { executeCommand(
100             new CutCommand(this, activeEditor)) }
101             cutButton.setCommand(cut)
102             shortcuts.onKeyPress("Ctrl+X", cut)
103
104            paste = function() { executeCommand(
105                new PasteCommand(this, activeEditor)) }
106                pasteButton.setCommand(paste)
107                shortcuts.onKeyPress("Ctrl+V", paste)
108
109                undo = function() { executeCommand(
110                    new UndoCommand(this, activeEditor)) }
111                    undoButton.setCommand(undo)
112                    shortcuts.onKeyPress("Ctrl+Z", undo)
113
114                    // 커맨드를 실행하여 기록에 추가해야 하는지 확인하세요.
115                    method executeCommand(command) is

```

```

116     if (command.execute)
117         history.push(command)
118
119     // 기록에서 가장 최근의 커맨드를 가져와서 그의 실행 취소 메서드를 실행하세요.
120     // 참고로 우리는 이 커맨드의 클래스를 알지 못한다는 사실에 유념하세요.
121     // 하지만 몰라도 상관없는데, 그 이유는 커맨드가 자신의 작업을 실행 취소하는
122     // 법을 알기 때문입니다.
123     method undo() is
124         command = history.pop()
125         if (command != null)
126             command.undo()

```

💡 적용

 **작업들로 객체를 매개변수화하려는 경우 커맨드 패턴을 사용하세요.**

 커맨드 패턴은 특정 메서드 호출을 독립실행형 객체로 전환할 수 있습니다. 이 변경을 통해 당신은 이제 커맨드들을 메서드 인수들로 전달하고, 이들을 다른 객체들의 내부에 저장하고, 런타임에 연결된 커맨드를 전환하는 등의 여러 흥미로운 작업을 진행할 수 있습니다.

예를 들어 당신은 콘텍스트 메뉴(상황에 맞는 메뉴)와 같은 그래픽 사용자 인터페이스 컴포넌트를 개발 중이고, 앱의 사용자들이 최종 사용자가 하나의 항목을 클릭했을 때 작업이 시작되는 메뉴 항목들을 설정할 수 있도록 만들고 싶어 합니다.

❖ 커맨드 패턴은 작업들의 실행을 예약하거나, 작업들을 대기열에 넣거나 작업들을 원격으로 실행하려는 경우에 사용하세요.

⚡ 다른 어느 객체와 마찬가지로 커맨드는 직렬화될 수 있습니다.
 직렬화는 파일이나 데이터베이스에 쉽게 쓸 수 있는 문자열로 변환하는 행위입니다. 나중에 이 문자열은 초기 커맨드 객체로 복원될 수 있습니다. 따라서 커맨드의 실행을 지연하고 예약할 수 있습니다. 또 같은 방식으로 네트워크를 통해 커맨드를 대기열에 추가하거나 로그(기록) 하거나 전송할 수 있습니다.

❖ 커맨드 패턴은 되돌릴 수 있는 작업을 구현하려고 할 때 사용하세요.

⚡ 실행 취소/다시 실행을 구현하는 방법에는 여러 가지가 있지만, 커맨드 패턴이 아마도 가장 많이 사용되는 패턴일 것입니다.

작업을 되돌리려면 수행한 작업의 기록을 구현해야 합니다. 커맨드 기록은 앱 상태의 관련 백업들과 함께 실행된 모든 커맨드 객체들을 포함하는 스택입니다.

이 메서드에는 두 가지 단점이 있습니다. 첫째, 앱 일부가 비공개일 수 있으므로 앱의 상태를 저장하는 것이 쉽지 않습니다. 이 문제는 메멘토 패턴으로 완화할 수 있습니다.

둘째, 상태 백업들은 상당히 많은 RAM을 소모할 수 있습니다. 따라서 때로는 대안적 구현에 의존할 수 있습니다. 예를 들어

과거 상태를 복원하는 대신 커맨드가 작업을 역으로 수행할 수 있습니다. 하지만 역으로 작업을 수행하는데도 대가가 있습니다. 구현하기 어렵거나 심지어 불가능할 수도 있습니다.

▣ 구현방법

1. 단일 실행 메서드로 커맨드 인터페이스를 선언하세요.
2. 요청들을 커맨드 인터페이스를 구현하는 구상 커맨드 클래스들로 추출하기 시작하세요. 각 클래스에는 실제 수신자 객체에 대한 참조와 함께 요청 인수들을 저장하기 위한 필드들의 집합이 있어야 합니다. 이러한 모든 값은 커맨드의 생성자를 통해 초기화되어야 합니다.
3. **발송자** 역할을 할 클래스들을 식별하세요. 이러한 클래스들에 커맨드들을 저장하기 위한 필드들을 추가하세요. 발송자들은 커맨드 인터페이스를 통해서만 커맨드들과 통신해야 합니다. 발송자들은 일반적으로 자체적으로 커맨드 객체들을 생성하지 않고 클라이언트 코드에서 가져옵니다.
4. 수신자에게 직접 요청을 보내는 대신 커맨드를 실행하도록 발송자들을 변경하세요.
5. 클라이언트는 다음 순서로 객체들을 초기화해야 합니다.
 - 수신자들을 만드세요.

- 커맨드들을 만들고 필요한 경우 수신자들과 연관시키세요.
- 발송자들을 만들고 특정 커맨드들과 연관시키세요.

⚠ 장단점

- ✓ 단일 책임 원칙. 작업을 호출하는 클래스들을 이러한 작업을 수행하는 클래스들로부터 분리할 수 있습니다.
- ✓ 개방/폐쇄 원칙. 기존 클라이언트 코드를 손상하지 않고 앱에 새 커맨드들을 도입할 수 있습니다.
- ✓ 실행 취소/다시 실행을 구현할 수 있습니다.
- ✓ 작업들의 자연된 실행을 구현할 수 있습니다.
- ✓ 간단한 커맨드들의 집합을 복잡한 커맨드로 조합할 수 있습니다.
- ✗ 발송자와 수신자 사이에 완전히 새로운 레이어를 도입하기 때문에 코드가 더 복잡해질 수 있습니다.

↔ 다른 패턴과의 관계

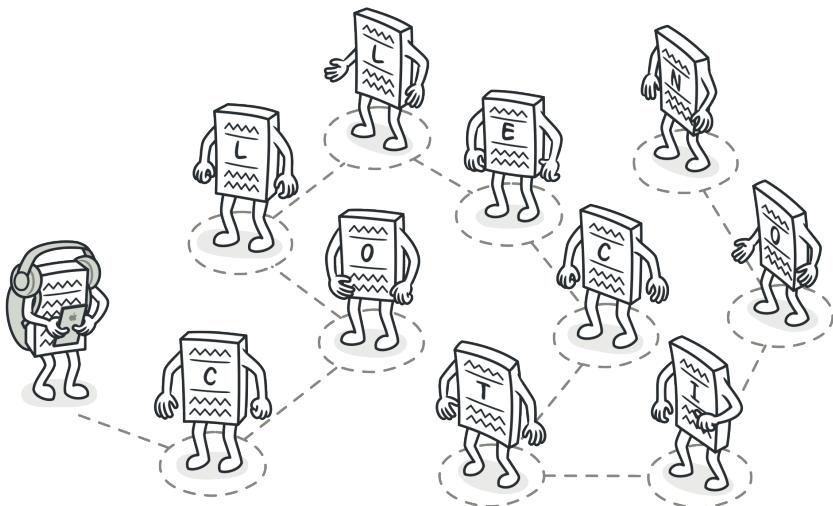
- 커맨드, 중재자, 옵서버 및 책임 연쇄 패턴은 요청의 발신자와 수신자를 연결하는 다양한 방법을 다룹니다.
 - 책임 연쇄 패턴은 잠재적 수신자의 동적 체인을 따라 수신자 중 하나에 의해 요청이 처리될 때까지 요청을 순차적으로 전달합니다.

- 커맨드 패턴은 발신자와 수신자 간의 단방향 연결을 설립합니다.
- 중재자 패턴은 발신자와 수신자 간의 직접 연결을 제거하여 그들이 중재자 객체를 통해 간접적으로 통신하도록 강제합니다.
- 옵서버 패턴은 수신자들이 요청들의 수신을 동적으로 구독 및 구독 취소할 수 있도록 합니다.
- 책임 연쇄 패턴의 핸들러들은 커맨드로 구현할 수 있습니다. 그러면 당신은 많은 다양한 작업을 같은 콘텍스트 객체에 대해 실행할 수 있으며, 해당 콘텍스트 객체는 요청의 역할을 합니다. 여기에서의 요청은 처리 메서드의 매개변수를 의미합니다.

그러나 요청 자체가 커맨드 객체인 다른 접근 방식이 있습니다. 이 접근 방식을 사용하면 당신은 같은 작업을 체인에 연결된 일련의 서로 다른 콘텍스트들에서 실행할 수 있습니다.

- 당신은 '실행 취소'를 구현할 때 커맨드와 메멘토 패턴을 함께 사용할 수 있습니다. 그러면 커맨드들은 대상 객체에 대해 다양한 작업을 수행하는 역할을 맡습니다. 반면, 메멘토들은 커맨드가 실행되기 직전에 해당 객체의 상태를 저장합니다.
- 커맨드와 전략 패턴은 비슷해 보일 수 있습니다. 왜냐하면 둘 다 어떤 작업으로 객체를 매개변수화하는 데 사용할 수 있기 때문입니다. 그러나 이 둘의 의도는 매우 다릅니다.

- 당신은 커맨드를 사용하여 모든 작업을 객체로 변환할 수 있습니다. 작업의 매개변수들은 해당 객체의 필드들이 됩니다. 이 변환은 작업의 실행을 연기하고, 해당 작업을 대기열에 넣고, 커맨드들의 기록을 저장한 후 해당 커맨드들을 원격 서비스에 보내는 등의 작업을 가능하게 합니다.
 - 반면에 전략 패턴은 일반적으로 같은 작업을 수행하는 다양한 방법을 설명하므로 단일 콘텍스트 클래스 내에서 이러한 알고리즘들을 교환할 수 있도록 합니다.
- 프로토타입은 커맨드 패턴의 복사본들을 기록에 저장해야 할 때 도움이 될 수 있습니다.
 - 비지터 패턴은 커맨드 패턴의 강력한 버전으로 취급할 수 있습니다. 비지터 패턴의 객체들은 다른 클래스들의 다양한 객체에 대한 작업을 실행할 수 있습니다.



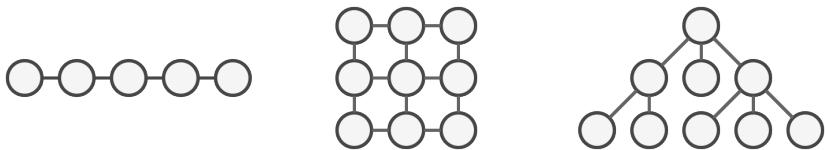
반복자 패턴

다음 이름으로도 불립니다: Iterator

반복자는 컬렉션의 요소들의 기본 표현(리스트, 스택, 트리 등)을 노출하지 않고 그들을 하나씩 순회할 수 있도록 하는 행동 디자인 패턴입니다.

(:) 문제

컬렉션은 프로그래밍에서 가장 많이 사용되는 데이터 유형 중 하나이긴 하지만, 객체 그룹의 단순한 컨테이너에 불과합니다.



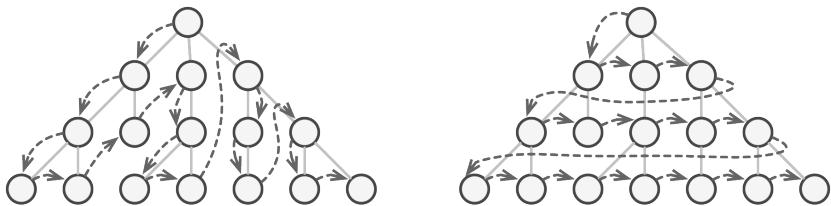
다양한 유형들의 컬렉션들.

대부분의 컬렉션들은 그들의 요소들을 간단한 리스트들에 저장하나, 그중 일부는 스택, 트리, 그래프 및 기타 복잡한 데이터 구조들을 기반으로 합니다.

그러나 컬렉션이 어떻게 구성되어 있는지를 떠나서, 컬렉션은 그 요소들에 접근할 수 있는 어떤 방법을 다른 코드에 제공해야 합니다. 그래야 다른 코드가 이 요소들을 사용할 수 있습니다. 같은 요소에 반복해서 접근하지 않고 컬렉션의 각 요소를 순회하는 방법이 있을 것입니다.

리스트로 된 컬렉션이 있다면 아주 쉽게 해결할 수 있을 겁니다. 모든 요소를 루프 처리하면 되니까요. 하지만 트리처럼 복잡한 데이터 구조의 요소들은 어떻게 순차적으로 순회해야 할까요? 예를 들어 어떤 날은 트리의 깊이를 우선으로 순회하는 것이 적절할지도 모릅니다. 하지만 그다음 날에는 너비를 우선으로

순회해야 할 수도 있습니다. 그리고 그다음 주에는 트리 요소들에 대한 임의 접근 등 다른 방식의 순회가 필요할지도 모릅니다.



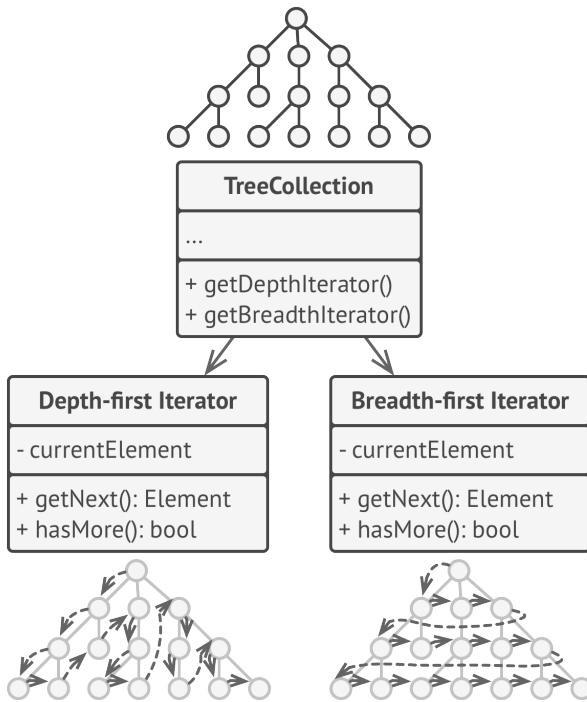
같은 컬렉션을 여러 가지 방법들로 순회할 수 있습니다.

현재 컬렉션의 주요 책임은 효율적인 데이터 저장이나, 컬렉션에 더 많은 순회 알고리즘들을 추가할수록 컬렉션의 주요 책임이 무엇인지 점점 명확해지지 않게 됩니다. 또한 일부 알고리즘들은 특정 앱에 맞게 조정되었을 수 있으므로 일반적인 컬렉션 클래스에 이들을 포함하는 것은 이상할 수 있습니다.

반면에 다양한 컬렉션들과 작동해야 하는 클라이언트 코드는 자신들의 요소가 어떻게 저장되는지 관심을 두지 않습니다. 하지만 컬렉션마다 그 요소들에 접근할 수 있도록 허용하는 방법이 다르므로, 당신은 코드를 특정한 컬렉션 클래스에 결합할 수밖에 없습니다.

⌚ 해결책

반복자 패턴의 주 아이디어는 컬렉션의 순회 동작을 반복자라는 별도의 객체로 추출하는 것입니다.



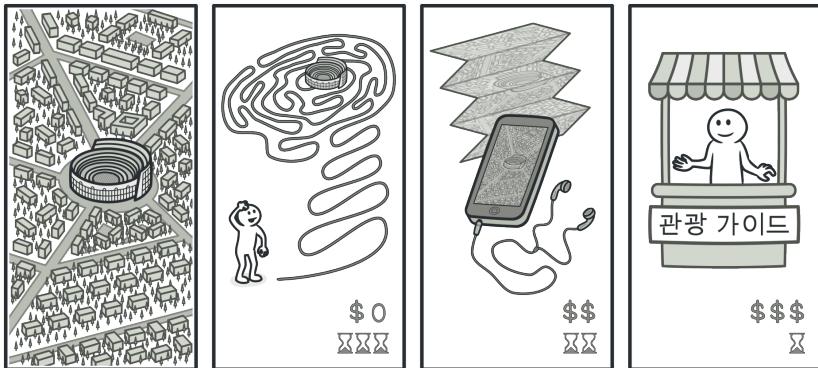
반복자들은 다양한 순회 알고리즘들을 구현합니다. 여러 반복자 객체들이 동시에 같은 컬렉션을 순회할 수 있습니다.

반복자 객체는 알고리즘 자체를 구현하는 것 외에도 모든 순회 세부 정보들(예: 현재 위치 및 남은 요소들의 수)을 캡슐화하며, 이 때문에 여러 반복자들이 서로 독립적으로 동시에 같은 컬렉션을 통과할 수 있습니다.

일반적으로 반복자들은 컬렉션의 요소들을 가져오기 위한 하나의 주 메서드를 제공합니다. 클라이언트는 이 메서드를 더 이상 아무것도 반환하지 않을 때까지 계속 실행할 수 있습니다. 이는 반복자가 모든 요소를 순회했음을 의미합니다.

모든 반복자들은 같은 인터페이스를 구현해야 합니다. 이렇게 하면 적절한 반복자가 있는 한 클라이언트 코드는 모든 컬렉션 유형들 및 순회 알고리즘들과 호환됩니다. 컬렉션을 순회하는 특별한 방법이 필요하면 컬렉션이나 클라이언트를 변경할 필요 없이 새 반복자 클래스를 만들기만 하면 됩니다.

🚗 실제상황 적용



도보로 로마를 탐험하는 다양한 방법들

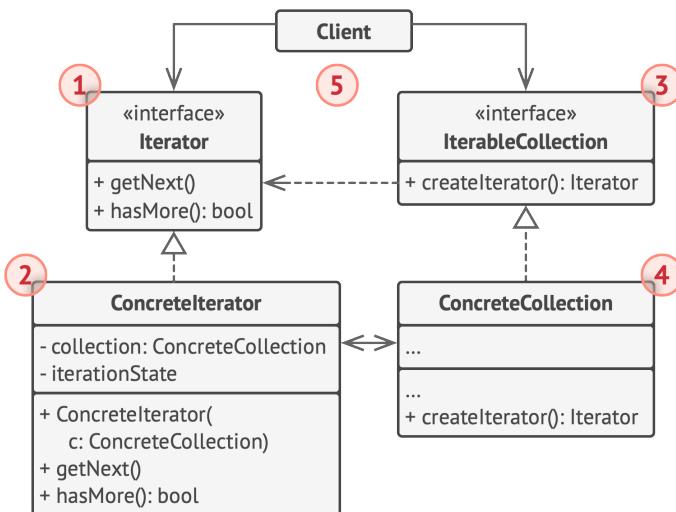
당신은 며칠 동안 로마를 방문하고 주요 명소들을 모두 방문할 계획을 세웠습니다. 그러나 막상 도착한 후 당신은 콜로세움조차 찾지 못하고 제자리를 맴도는 데 많은 시간을 허비할 수 있습니다.

위 사례의 대안으로 스마트폰용 가상 가이드 앱을 구매하여 내비게이션에 사용할 수 있습니다. 이 앱은 똑똑하고 저렴하며 실제 가이드와 달리 원하는 만큼 흥미로운 장소에 머물 수 있도록 합니다.

세 번째 대안은 조금 더 비싸더라도 도시를 잘 아는 현지 가이드를 고용하는 것입니다. 현지 가이드는 당신의 취향에 맞게 여행을 조정하고 모든 명소를 보여주며 흥미진진한 이야기들을 많이 알려줄 수 있습니다. 이 대안을 선택하면 재미는 있겠지만, 더 많은 비용이 들게 될 것입니다.

이 모든 대안들은 (예: 당신이 무작위로 생각해 낸 방향들, 스마트폰 내비게이터, 현지 가이드) 로마의 많은 관광명소에 대해 반복자들로 작동합니다.

구조



1. **반복자** 인터페이스는 컬렉션의 순회에 필요한 작업들(예: 다음 요소 가져오기, 현재 위치 가져오기, 반복자 다시 시작 등)을 선언합니다.

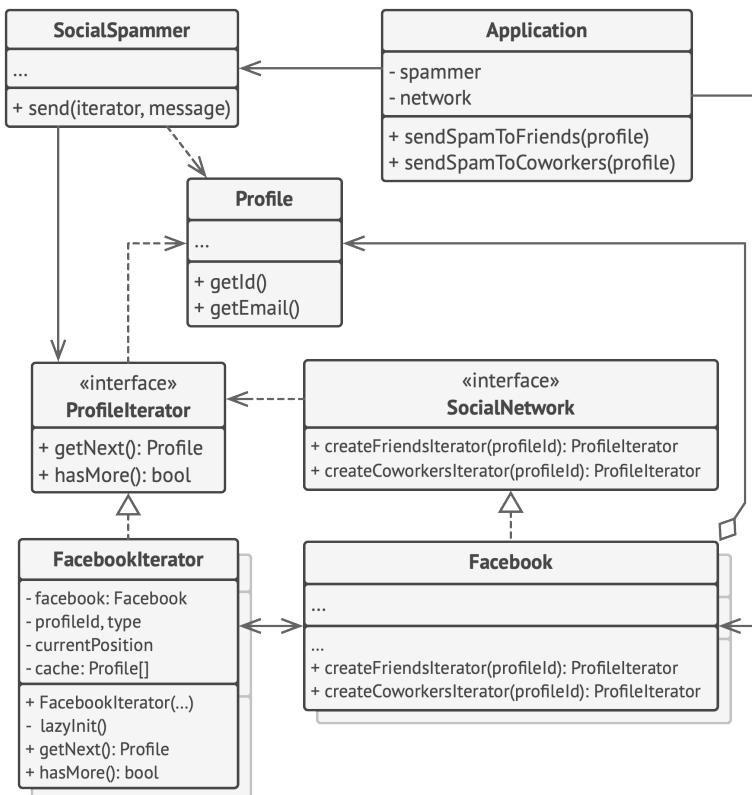
2. **구상 반복자들은** 컬렉션 순회를 위한 특정 알고리즘들을 구현합니다. 반복자 객체는 순회의 진행 상황을 자체적으로 추적해야 합니다. 이는 여러 반복자들이 같은 컬렉션을 서로 독립적으로 순회할 수 있도록 합니다.
3. **컬렉션 인터페이스는** 컬렉션과 호환되는 반복자들을 가져오기 위한 하나 이상의 메서드들을 선언합니다. 참고로 메서드들의 반환 유형은 반복자 인터페이스의 유형으로 선언되어야 합니다. 그래야 구상 컬렉션들이 다양한 유형의 반복자들을 반환할 수 있기 때문입니다.
4. **구상 컬렉션들은** 클라이언트가 요청할 때마다 특정 구상 반복자 클래스의 새 인스턴스들을 반환합니다. 당신은 컬렉션의 나머지 코드가 어디에 있는지 궁금하실 수도 있습니다. 걱정하지 마세요, 같은 클래스에 있을 것입니다. 나머지 코드가 어디에 있는지와 같은 세부 사항들은 실제 패턴에 중요하지 않으므로 생략하기로 했습니다.
5. **클라이언트는** 반복자들과 컬렉션들의 인터페이스를 통해 그들과 함께 작동합니다. 이렇게 하면 클라이언트가 구상 클래스들에 결합하지 않으므로 같은 클라이언트 코드로 다양한 컬렉션들과 반복자들을 사용할 수 있도록 합니다.

일반적으로 클라이언트들은 자체적으로 반복자들을 생성하지 않고 대신 컬렉션들에서 가져옵니다. 그러나 어떤 경우에는 (예를

들어 클라이언트가 자체 특수 반복자를 정의할 때) 클라이언트가 반복자를 직접 만들 수 있습니다.

의사코드

이 예시에서 **반복자** 패턴은 페이스북의 소셜 그래프에 대한 접근을 캡슐화하는 특별한 종류의 컬렉션을 순회하는 데 사용합니다. 이 컬렉션은 다양한 방식으로 프로필들을 순회할 수 있는 여러 반복자들을 제공합니다.



소셜 프로필들의 순회 예시.

`Friends` (친구들) 반복자는 주어진 프로필의 친구들을 탐색하는 데 사용될 수 있습니다. `Colleagues` (동료들) 반복자는 프로필 주인과 같은 회사에서 일하지 않는 친구들을 제외한 후 같은 작업을 수행합니다. 두 반복자 모두 인증 및 REST 요청 전송과 같은 구현 세부 사항들을 자세히 살펴보지 않고도 클라이언트들이 프로필들을 가져올 수 있도록 하는 공통 인터페이스를 구현합니다.

클라이언트 코드는 인터페이스들을 통해서만 컬렉션들 및 반복자들과 작업하기 때문에 구상 클래스들과 결합하지 않습니다. 앱을 새로운 소셜 네트워크에 연결하기로 했다면 기존 코드를 변경하지 않고 새로운 컬렉션 및 반복자 클래스들을 제공하기만 하면 됩니다.

```

1 // 컬렉션 인터페이스는 반복자들을 생성하기 위한 팩토리 메서드를 선언해야 합니다.
2 // 프로그램에서 사용할 수 있는 다양한 종류의 순회가 있는 경우 여러 메서드를 선언할
3 // 수 있습니다.
4 interface SocialNetwork is
5     method createFriendsIterator(profileId):ProfileIterator
6     method createCoworkersIterator(profileId):ProfileIterator
7
8
9 // 각 구상 컬렉션은 자신이 반환하는 구상 반복자 클래스들의 집합에 연결됩니다.
10 // 하지만 이러한 메서드들의 시그니처는 반복자 인터페이스를 반환하기 때문에
11 // 클라이언트는 연결되지 않습니다.
12 class Facebook implements SocialNetwork is
13     // ...컬렉션 코드의 대부분은 여기에 포함되어야 합니다...
14

```

```

15 // 반복자 생성 코드.
16 method createFriendsIterator(profileId) is
17     return new FacebookIterator(this, profileId, "friends")
18 method createCoworkersIterator(profileId) is
19     return new FacebookIterator(this, profileId, "coworkers")
20
21
22 // 모든 반복자에 대한 공통 인터페이스.
23 interface ProfileIterator is
24     method getNext():Profile
25     method hasMore():bool
26
27
28 // 구상 반복자 클래스.
29 class FacebookIterator implements ProfileIterator is
30     // 반복자는 순회하는 컬렉션에 대한 참조가 필요합니다.
31     private field facebook: Facebook
32     private field profileId, type: string
33
34     // 반복자 객체는 다른 반복자들과 별도로 컬렉션을 순회합니다. 따라서 반복자
35     // 상태를 저장해야 합니다.
36     private field currentPosition
37     private field cache: array of Profile
38
39     constructor FacebookIterator(facebook, profileId, type) is
40         this.facebook = facebook
41         this.profileId = profileId
42         this.type = type
43
44     private method lazyInit() is
45         if (cache == null)
46             cache = facebook.socialGraphRequest(profileId, type)

```

```

47
48 // 각 구상 반복자 클래스는 공통 반복자 인터페이스를 자체적으로 구현합니다.
49 method getNext() is
50     if (hasMore())
51         result = cache[currentPosition]
52         currentPosition++
53         return result
54
55 method hasMore() is
56     lazyInit()
57     return currentPosition < cache.length
58
59
60 // 유용한 요령이 하나 더 있습니다. 전체 컬렉션에 대한 접근 권한을 클라이언트
61 // 클래스에 부여하는 대신 반복자를 클라이언트 클래스에 전달하는 것입니다. 그러면
62 // 컬렉션이 클라이언트에 노출되지 않습니다.
63 //
64 // 장점도 하나 더 있습니다. 클라이언트에 다른 반복자를 전달하여 런타임 때
65 // 클라이언트가 컬렉션과 작동하는 방식을 변경할 수 있습니다. 이것은 클라이언트
66 // 코드가 구상 반복자 클래스들에 결합되어 있지 않기 때문에 가능합니다.
67 class SocialSpammer is
68     method send(iterator: ProfileIterator, message: string) is
69         while (iterator.hasMore())
70             profile = iterator.getNext()
71             System.sendEmail(profile.getEmail(), message)
72
73
74 // 앱 클래스는 컬렉션들과 반복자들을 설정한 다음 클라이언트 코드에 전달합니다.
75 class Application is
76     field network: SocialNetwork
77     field spammer: SocialSpammer
78

```

```

79  method config() is
80      if working with Facebook
81          this.network = new Facebook()
82      if working with LinkedIn
83          this.network = new LinkedIn()
84      this.spammer = new SocialSpammer()
85
86  method sendSpamToFriends(profile) is
87      iterator = network.createFriendsIterator(profile.getId())
88      spammer.send(iterator, "Very important message")
89
90  method sendSpamToCoworkers(profile) is
91      iterator = network.createCoworkersIterator(profile.getId())
92      spammer.send(iterator, "Very important message")

```

💡 적용

💡 반복자 패턴은 당신의 컬렉션이 내부에 복잡한 데이터 구조가 있지만 이 구조의 복잡성을 보안이나 편의상의 이유로 클라이언트들로부터 숨기고 싶을 때 사용하세요.

⚡ 반복자는 복잡한 데이터 구조와 작업 시의 세부 사항을 캡슐화하여 클라이언트에 컬렉션 요소들에 접근할 수 있는 몇 가지 간단한 메서드들을 제공합니다. 이 접근 방식은 클라이언트에게 매우 편리합니다. 또 클라이언트가 컬렉션과 직접 작동할 때 클라이언트가 수행할 수 있는 부주의하거나 악의적인 행동들로부터 컬렉션을 보호합니다.

 **반복자 패턴을 사용하여 당신의 앱 전체에서 순회 코드의 중복을 줄이세요.**

 사소하지 않은 순회 알고리즘들의 코드는 부피가 매우 큰 경향이 있습니다. 이 코드들이 앱의 비즈니스 로직 내에 배치되면 원래 코드의 책임이 무엇인지 모호해질 수 있으며 코드의 유지관리가 더 어려워질 수 있습니다. 순회 코드를 지정된 반복자들로 이동하면 당신의 앱의 코드가 더 간결하고 깔끔해질 수 있습니다.

 **반복자 패턴은 코드가 다른 데이터 구조들을 순회할 수 있기를 원할 때 또는 이러한 구조들의 유형을 미리 알 수 없을 때 사용하세요.**

 이 패턴은 컬렉션들과 반복자들 모두에 몇 개의 일반 인터페이스들을 제공합니다. 당신의 코드가 이러한 인터페이스들을 사용한다는 점을 고려할 때, 이러한 인터페이스들은 그들을 구현하는 다양한 컬렉션들 및 반복자들을 전달받아도 여전히 작동합니다.

구현방법

1. 반복자 인터페이스를 선언하세요. 이 인터페이스는 최소한 컬렉션에서 다음 요소를 가져오는 메서드가 있어야 하며, 또 편의를 위해 여기에 몇 가지 다른 메서드들도 (예: 전 요소를

가져오는, 현재 위치를 추적하는, 그리고 반복자의 순회의 끝을 확인하는 메서드들) 추가할 수 있습니다.

2. 컬렉션 인터페이스를 선언하고 반복자를 가져오는 메서드를 설명하세요. 컬렉션 인터페이스의 반환 유형은 반복자 인터페이스의 유형과 같아야 합니다. 뚜렷하게 다른 여러 개의 반복자들의 그룹을 가질 계획이라면 유사한 메서드들을 선언할 수 있습니다.
3. 반복자들이 순회하게 할 수 있도록 하고 싶은 컬렉션들에 대한 구상 반복자 클래스들을 구현하세요. 반복자 객체는 단일 컬렉션 인스턴스와 반드시 연결되어야 합니다. 일반적으로 이러한 연결은 반복자의 생성자를 통해 맺어집니다.
4. 당신의 컬렉션 클래스들에서 컬렉션 인터페이스를 구현하세요. 그 주된 목적은 클라이언트에 특정 컬렉션 클래스에 맞게 조정된 반복자들을 생성하기 위한 바로 가기를 제공하는 것입니다. 컬렉션 객체는 반복자의 생성자에 자신을 전달해야 하며 이 둘 사이에 연결을 맺어야 합니다.
5. 클라이언트 코드를 살펴보면서 반복자들을 사용하여 모든 컬렉션 순회 코드들을 교체하세요. 클라이언트는 컬렉션 요소들을 순회해야 할 때마다 새 반복자 객체를 가져옵니다.

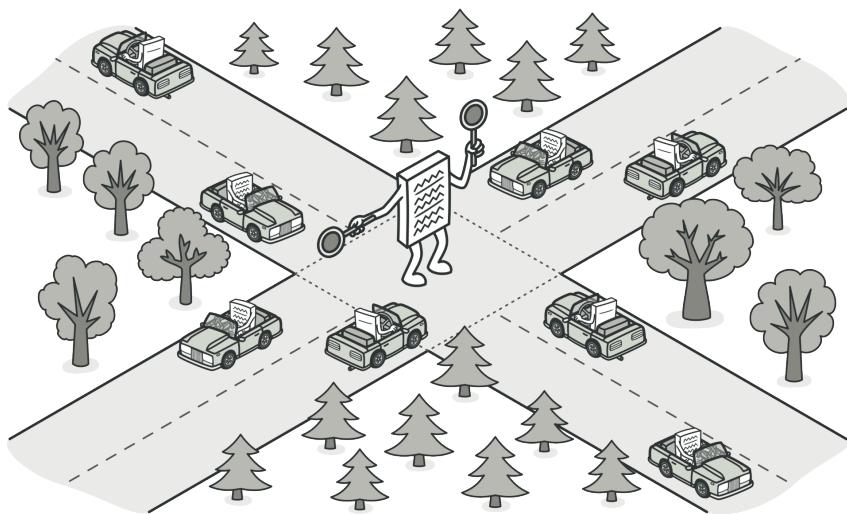
⚠️ 장단점

- ✓ 단일 책임 원칙. 부피가 큰 순회 알고리즘들을 별도의 클래스들로 추출하여 클라이언트 코드와 컬렉션들을 정돈할 수 있습니다.
- ✓ 개방/폐쇄 원칙. 새로운 유형의 컬렉션들과 반복자들을 구현할 수 있으며 이들을 아무것도 훼손하지 않은 채 기존의 코드에 전달할 수 있습니다.
- ✓ 당신은 이제 같은 컬렉션을 병렬로 순회할 수 있습니다. 왜냐하면 각 반복자 객체에는 자신의 고유한 순회 상태가 포함되어 있기 때문입니다.
- ✓ 같은 이유로 당신은 순회를 지연하고 필요할 때 계속할 수 있습니다.
- ✗ 당신의 앱이 단순한 컬렉션들과만 작동하는 경우 반복자 패턴을 적용하는 것은 과도할 수 있습니다.
- ✗ 반복자를 사용하는 것은 일부 특수 컬렉션들의 요소들을 직접 탐색하는 것보다 덜 효율적일 수 있습니다.

↔ 다른 패턴과의 관계

- 당신은 반복자들을 사용하여 **복합체** 패턴 트리들을 순회할 수 있습니다.

- **팩토리 메서드**를 **반복자**와 함께 사용하여 컬렉션 자식 클래스들이 해당 컬렉션들과 호환되는 다양한 유형의 반복자들을 반환하도록 할 수 있습니다.
- **메멘토** 패턴을 **반복자** 패턴과 함께 사용하여 현재 순회 상태를 포착하고 필요한 경우 롤백할 수 있습니다.
- **비지터** 패턴과 **반복자** 패턴을 함께 사용해 복잡한 데이터 구조를 순회하여 해당 구조의 요소들의 클래스들이 모두 다르더라도 이러한 요소들에 대해 어떤 작업을 실행할 수 있습니다.



중재자 패턴

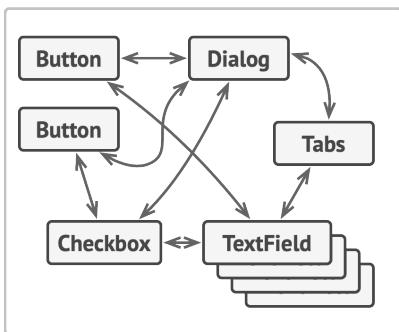
다음 이름으로도 불립니다: 중개인, 컨트롤러, Mediator

중재자는 객체 간의 혼란스러운 의존 관계들을 줄일 수 있는 행동 디자인 패턴입니다. 이 패턴은 객체 간의 직접 통신을 제한하고 중재자 객체를 통해서만 협력하도록 합니다.

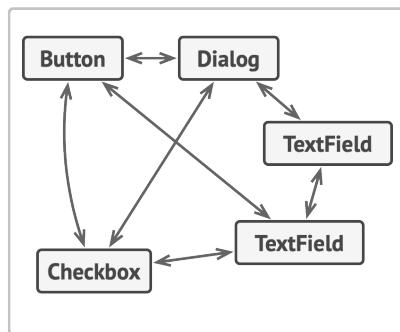
(:) 문제

고객들의 프로필을 만들고 편집하기 위한 대화 상자가 있다고 가정해 봅시다. 이 대화 상자는 텍스트 필드, 체크 상자, 버튼 등과 같은 다양한 양식 컨트롤들로 구성됩니다.

프로필 대화상자



로그인 대화상자



앱이 발전함에 따라 사용자 인터페이스 요소 간의 관계가 혼란스러워질 수 있습니다.

일부 양식 요소들은 다른 요소들과 상호 작용할 수 있습니다. 예를 들어, '저는 개가 있습니다' 확인란을 선택하면 개의 이름을 입력하기 위한 숨겨진 텍스트 필드가 나타날 수 있습니다. 또 다른 예시로 데이터를 저장하기 전에 모든 필드의 값들을 검증해야 하는 제출 버튼이 있습니다.



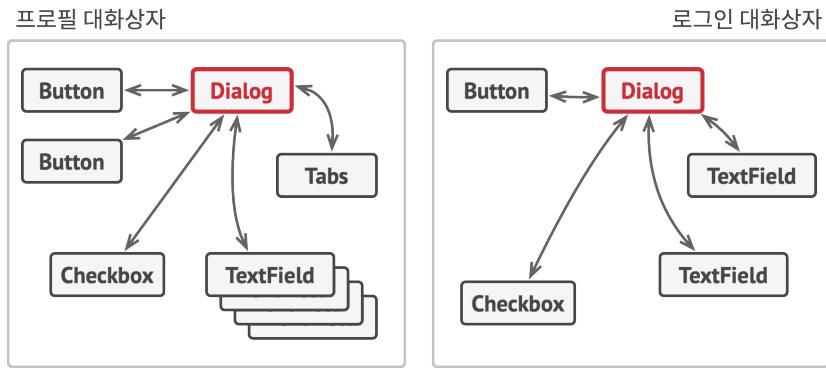
요소들은 다른 요소들과 많은 관계를 맺을 수 있습니다. 따라서 일부 요소들을 변경하면 다른 요소들에 영향을 줄 수 있습니다.

이 논리를 양식 요소들의 코드 내에서 직접 구현하면 이러한 요소들의 클래스들을 앱의 다른 양식들에서 재사용하기가 훨씬 더 어려워집니다. 예를 들어, 다른 양식 내에서는 위에 언급한 개 관련 확인란 클래스를 사용할 수 없습니다. 왜냐하면 기존 클래스가 개의 이름을 입력하기 위한 텍스트 필드와 결합되어 있기 때문입니다. 이 경우 프로필 양식 렌더링과 관련된 클래스들을 전부 사용하거나 아니면 아예 사용하지 말아야 합니다.

呵呵 해결책

중재자 패턴은 서로 독립적으로 작동해야 하는 컴포넌트 간의 모든 직접 통신을 중단한 후, 대신 이러한 컴포넌트들은 호출들을 적절한 컴포넌트들로 리다이렉션하는 특수 중재자 객체를 호출하여 간접적으로 협력하게 하라고 제안합니다. 그러면 컴포넌트들은 수십 개의 동료 컴포넌트들과 결합되는 대신 단일 중재자 클래스에만 의존합니다.

위 프로필 편집 양식 예시에서는 대화 상자 클래스 자체가 중재자 역할을 할 수 있습니다. 아마도 대화 상자 클래스는 이미 자신의 모든 하위 요소들을 인식하고 있으므로 새로운 의존관계들을 도입할 필요가 없을 것입니다.



UI 요소들은 다른 UI 요소들과 중재자 객체를 통해 간접적으로
통신해야 합니다.

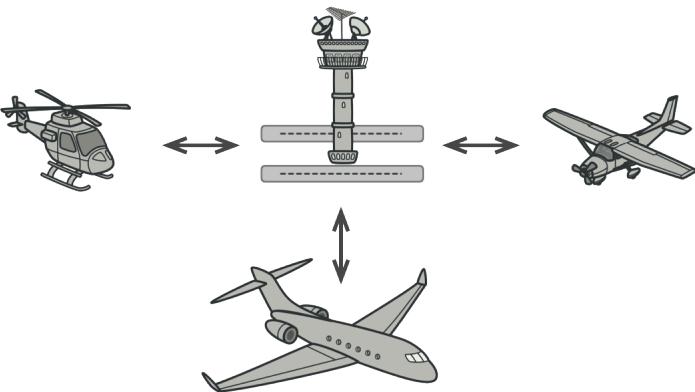
가장 중요한 변경들은 실제 양식 요소들에 적용됩니다. 제출 버튼을 살펴보면 이전에는 사용자가 이 버튼을 클릭할 때마다 버튼은 모든 개별 양식 요소들의 값들을 검증해야 했습니다. 이제 제출 버튼이 해야 할 유일한 일은 클릭을 대화 상자에 알리는 것 하나입니다. 이 알림을 받으면 대화 상자는 스스로 검증을 수행하거나 개별 요소들에게 작업을 전달합니다. 따라서 버튼은 여러 개의 양식 요소들에 연결되는 대신 대화 상자 클래스에만 의존하게 됩니다.

여기서 더 나아가 모든 유형의 대화 상자에서 공통 인터페이스를 추출하여 의존성을 더욱 느슨하게 만들 수 있습니다. 이

인터페이스는 모든 양식 요소가 해당 요소들에 발생하는 일(이벤트)들을 대화 상자에 알리는 데 사용할 수 있는 알림 메서드를 선언합니다. 이렇게 하면 제출 버튼은 이제 해당 인터페이스를 구현하는 모든 대화 상자들과 작업할 수 있습니다.

그렇게 하면, 중재자 패턴은 단일 중재자 객체 내부의 다양한 객체 간의 복잡한 관계망을 캡슐화할 수 있도록 합니다. 클래스의 의존관계들이 적을수록 해당 클래스를 수정, 확장 또는 재사용하기가 더 쉬워집니다.

🚗 실제상황 적용



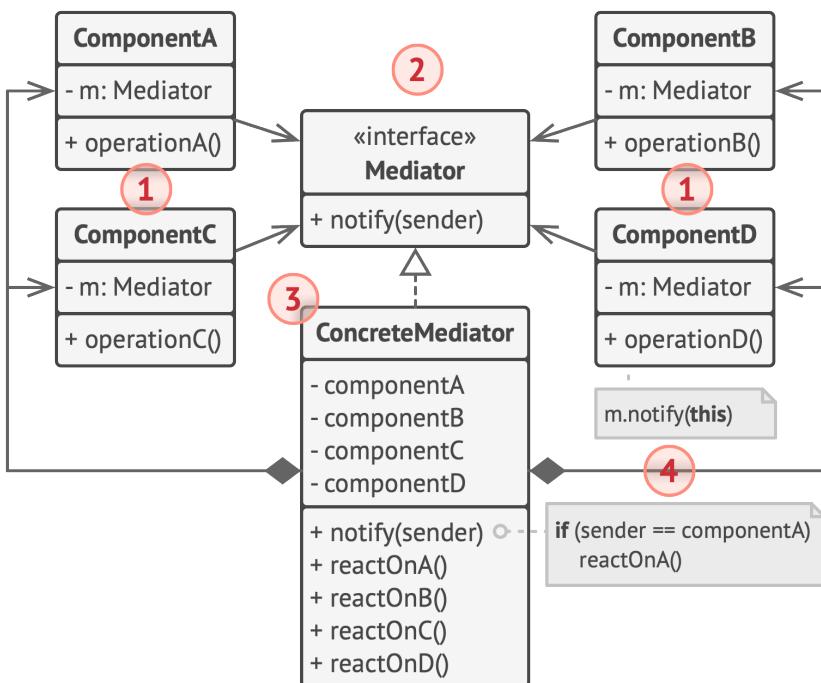
항공기 조종사들은 다음에 누가 비행기를 착륙시킬지를 결정할 때 서로 직접 대화하지 않습니다. 모든 통신은 비행기 관제탑을 통해 이루어집니다.

공항 관제 구역으로 들어오거나 그곳을 떠나는 항공기의 조종사들은 서로 직접 통신하지 않습니다. 대신, 그들은 높은 타워에 앉아서 일하는 항공 교통 관제사와 통신합니다. 항공 교통

관제사가 없다면 조종사들은 공항 근처의 모든 비행기의 존재 여부를 인식하고 수십 명의 다른 조종사들로 구성된 위원회와 착륙 우선순위를 논의해야 합니다. 그러면 비행기 충돌 횟수는 아마도 하늘로 치솟을 것입니다.

관제탑은 전체 비행을 관할하지 않습니다. 다만 관련되는 비행기의 수가 조종사에게는 너무 많을 수 있기에 공항 터미널 지역에서만 제약들을 강제하기 위해 존재합니다.

구조



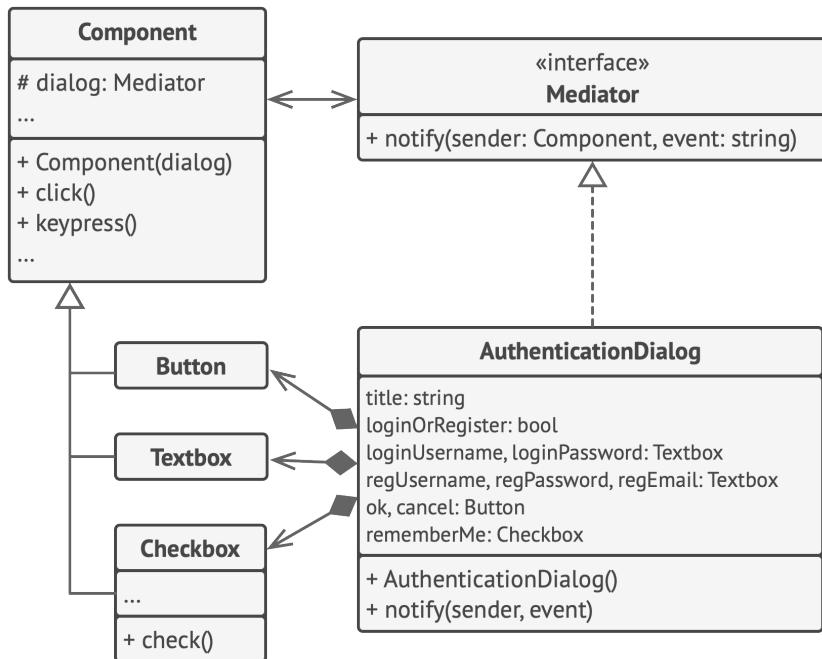
1. **컴포넌트들은** 어떤 비즈니스 로직을 포함한 다양한 클래스들입니다. 각 컴포넌트에는 중재자에 대한 참조가 있는데, 이 중재자는 중재자 인터페이스의 유형으로 선언됩니다. 컴포넌트는 중재자의 실제 클래스를 인식하지 못하므로 컴포넌트를 다른 중재자에 연결하여 다른 프로그램에서 재사용할 수 있습니다.
2. **중재자 인터페이스는** 일반적으로 단일 알림 메서드만을 포함하는 컴포넌트들과의 통신 메서드들을 선언합니다. 컴포넌트들은 자체 객체들을 포함하여 모든 콘텍스트를 이 메서드의 인수로 전달할 수 있지만 이는 수신자 컴포넌트와 발송자 클래스 간의 결합이 발생하지 않는 방식으로만 가능합니다.
3. **구상 중재자들은** 다양한 컴포넌트 간의 관계를 캡슐화합니다. 구상 중재자들은 자신이 관리하는 모든 컴포넌트에 대한 참조를 유지하고 때로는 그들의 수명 주기를 관리하기도 합니다.
4. 컴포넌트들은 다른 컴포넌트들을 인식하지 않아야 합니다. 컴포넌트 내에서 또는 컴포넌트에 중요한 일이 발생하면, 컴포넌트는 이를 중재자에게만 알려야 합니다. 중재자는 알림을 받으면 발송자를 쉽게 식별할 수 있으며, 이는 응답으로 어떤 컴포넌트가 작동되어야 하는지 결정하기에 충분할 수 있습니다.

컴포넌트의 관점에서는 모든 것들이 블랙박스들(기능은 알지만, 작동 원리를 이해할 수 없는 복잡한 기계나 시스템)처럼 보입니다.

발송자는 누가 요청을 처리할지 모르고, 수신자는 누가 처음에 요청을 보냈는지를 모릅니다.

의사코드

이 예시에서 **중재자** 패턴은 버튼들, 확인란들 및 텍스트 레이블들과 같은 다양한 UI 클래스 간의 상호 의존성을 제거하는데 도움이 됩니다.



UI 대화 상자 클래스들의 구조.

사용자에 의해 작동된 요소는 다른 요소들과 직접 통신하지 않습니다. 대신 이 요소는 중재자에게 이 이벤트(사건)에 대해

알리고 중재자에게 해당 알림과 함께 컨텍스트 정보를 전달합니다.

이 예시에서는 인증 대화 상자 전체가 중재자의 역할을 합니다. 이것은 구상 요소들이 어떻게 협력해야 하는지 알고 있으며 그들의 간접적인 의사소통을 촉진합니다. 이벤트에 대한 알림을 받으면 대화 상자는 이벤트를 처리해야 하는 요소를 결정하고 그 결정에 따라 호출을 리다이렉션합니다.

```

1 // 중재자 인터페이스는 컴포넌트들에서 사용하는 메서드를 선언하여 다양한 이벤트를
2 // 중재자에게 알립니다. 중재자는 이러한 이벤트에 반응해 실행을 다른 컴포넌트들에게
3 // 전달할 수 있습니다.
4 interface Mediator is
5   method notify(sender: Component, event: string)
6
7
8 // 구상 중재자 클래스. 개별 컴포넌트들의 얹히고설킨 연결들이 풀리고 중재자로
9 // 이동되었습니다.
10 class AuthenticationDialog implements Mediator is
11   private field title: string
12   private field loginOrRegisterChkBx: Checkbox
13   private field loginUsername, loginPassword: Textbox
14   private field registrationUsername, registrationPassword,
15     registrationEmail: Textbox
16   private field okBtn, cancelBtn: Button
17
18 constructor AuthenticationDialog() is
19   // 연결을 설립하기 위해 현재 중재자를 컴포넌트 객체들의 생성자들에
20   // 전달하여 모든 컴포넌트 객체들을 생성하세요.

```

```

21
22 // 컴포넌트에 무슨 일어나면, 중재자에게 알립니다. 알림을 받으면 중재자는
23 // 자체적으로 알림을 처리하거나 요청을 다른 컴포넌트에 전달할 수 있습니다.
24 method notify(sender, event) is
25   if (sender == loginOrRegisterChkBx and event == "check")
26     if (loginOrRegisterChkBx.checked)
27       title = "Log in"
28       // 1. 로그인 양식 컴포넌트들을 표시하세요.
29       // 2. 등록 양식 컴포넌트들을 표시하세요.
30   else
31     title = "Register"
32     // 1. 등록 양식 컴포넌트들을 표시하세요.
33     // 2. 로그인 양식 컴포넌트들을 숨기세요.
34
35   if (sender == okBtn && event == "click")
36     if (loginOrRegister.checked)
37       // 로그인 자격 증명을 사용하여 사용자를 찾아보세요.
38       if (!found)
39         // 로그인 필드 위에 오류 메시지를 표시하세요.
40   else
41     // 1. 등록 필드의 데이터를 사용하여 사용자 계정을 만드세요.
42     // 2. 해당 사용자를 로그인시키세요.
43     // ...
44
45
46 // 컴포넌트들은 중재자 인터페이스를 사용하여 중재자와 통신합니다. 덕분에 컴포넌트들을
47 // 다른 중재자 객체들과 연결하여 다른 콘텍스트에서 같은 컴포넌트들을 사용할 수
48 // 있습니다.
49 class Component is
50   field dialog: Mediator
51
52   constructor Component(dialog) is

```

```

53     this.dialog = dialog
54
55     method click() is
56         dialog.notify(this, "click")
57
58     method keypress() is
59         dialog.notify(this, "keypress")
60
61 // 구상 컴포넌트들은 서로 통신하지 않습니다. 그들은 하나의 통신 채널만 가지고
62 // 있으며, 이 채널을 통해 중재자에게 알림들을 보냅니다.
63 class Button extends Component is
64     // ...
65
66 class Textbox extends Component is
67     // ...
68
69 class Checkbox extends Component is
70     method check() is
71         dialog.notify(this, "check")
72     // ...

```

💡 적용

💡 중재자 패턴은 일부 클래스들이 다른 클래스들과 단단하게 결합하여 변경하기 어려울 때 사용하세요.

⚡ 중재자 패턴을 사용하면 특정 컴포넌트에 대한 모든 변경을 나머지 컴포넌트들로부터 고립하며 클래스 간의 모든 관계들을 별도의 클래스로 추출할 수 있습니다.

☞ 이 패턴은 타 컴포넌트들에 너무 의존하기 때문에 다른 프로그램에서 컴포넌트를 재사용할 수 없는 경우 사용하세요.

↳ 중재자 패턴을 적용하면, 그 후 개별 컴포넌트들은 다른 컴포넌트들을 인식하지 못합니다. 또, 비록 간접적이기는 하지만 컴포넌트들은 중재자 객체를 통해 여전히 서로 통신할 수 있습니다. 다른 앱에서 컴포넌트를 재사용하려면 그 앱에 새 중재자 클래스를 제공해야 합니다.

☞ 중재자 패턴은 몇 가지 기본 행동을 다양한 콘텍스트들에서 재사용하기 위해 수많은 컴포넌트 자식 클래스들을 만들고 있는 스스로를 발견했을 때 사용하세요.

↳ 컴포넌트들 간의 모든 관계들이 중재자 내에 포함되어 있으므로 컴포넌트들 자체를 변경할 필요 없이 새 중재자 클래스들을 도입하여 이러한 컴포넌트들이 협업할 수 있는 완전히 새로운 방법들을 쉽게 정의할 수 있습니다.

▣ 구현방법

1. 더 독립적으로 만들었을 때 클래스의 유지 관리가 더 쉬워지거나 재사용이 더 간편해지는 등의 이점이 있는 단단히 결합된 클래스들을 식별하세요.

2. 중재자 인터페이스를 선언하고 중재자와 다양한 컴포넌트 간의 원하는 통신 프로토콜을 설명하세요. 대부분의 경우 컴포넌트들에서 알림을 수신하는 단일 메서드로 충분합니다.

이 인터페이스는 다른 콘텍스트들에서 컴포넌트 클래스들을 재사용하고자 할 때 매우 중요합니다. 컴포넌트가 일반 인터페이스를 통해 중재자와 함께 작동하는 한 컴포넌트를 중재자의 다른 구현과 연결할 수 있습니다.

3. 구상 중재자 클래스를 구현하세요. 모든 컴포넌트에 대한 참조를 중재자 내부에 저장하는 것을 고려해보세요. 그렇게 하면 중재자의 메서드에서 어떤 컴포넌트라도 호출할 수 있습니다.
4. 더 나아가 중재자가 컴포넌트 객체들의 생성 및 파괴를 담당하도록 할 수 있으며, 그러면 중재자는 **팩토리** 또는 **퍼사드**와 유사할 수 있습니다.
5. 컴포넌트들은 중재자 객체에 대한 참조를 저장해야 합니다. 이 연결은 일반적으로 컴포넌트의 생성자에서 설정되며 중재자 객체가 인수로 전달됩니다.
6. 다른 컴포넌트들의 메서드 대신 중재자의 알림 메서드를 호출하도록 컴포넌트의 코드를 변경하세요. 그 후 다른 컴포넌트들을 호출하는 것과 관련된 코드를 중재자 클래스 안으로 추출하세요. 이 코드는 중재자가 해당 컴포넌트에서 알림들을 받을 때마다 실행하세요.

⚠️ 장단점

- ✓ 단일 책임 원칙. 다양한 컴포넌트 간의 통신을 한곳으로 추출하여 코드를 이해하고 유지 관리하기 쉽게 만들 수 있습니다.
- ✓ 개방/폐쇄 원칙. 실제 컴포넌트들을 변경하지 않고도 새로운 중재자들을 도입할 수 있습니다.
- ✓ 프로그램의 다양한 컴포넌트 간의 결합도를 줄일 수 있습니다.
- ✓ 개별 컴포넌트들을 더 쉽게 재사용할 수 있습니다.
- ✗ 중재자는 전지전능한 객체로 발전할지도 모릅니다.

↔ 다른 패턴과의 관계

- 커맨드, 중재자, 옵서버 및 책임 연쇄 패턴은 요청의 발신자와 수신자를 연결하는 다양한 방법을 다룹니다.
 - 책임 연쇄 패턴은 잠재적 수신자의 동적 체인을 따라 수신자 중 하나에 의해 요청이 처리될 때까지 요청을 순차적으로 전달합니다.
 - 커맨드 패턴은 발신자와 수신자 간의 단방향 연결을 설립합니다.
 - 중재자 패턴은 발신자와 수신자 간의 직접 연결을 제거하여 그들이 중재자 객체를 통해 간접적으로 통신하도록 강제합니다.

- 옵서버 패턴은 수신자들이 요청들의 수신을 동적으로 구독 및 구독 취소할 수 있도록 합니다.
- **중재자와 퍼사드** 패턴은 비슷한 역할을 합니다. 둘 다 밀접하게 결합된 많은 클래스 간의 협업을 구성하려고 합니다.
 - 퍼사드 패턴은 객체들의 하위 시스템에 대한 단순화된 인터페이스를 정의하지만 새로운 기능을 도입하지는 않습니다. 하위 시스템 자체는 퍼사드를 인식하지 못하며, 하위 시스템 내의 객체들은 서로 직접 통신할 수 있습니다.
 - 중재자는 시스템 컴포넌트 간의 통신을 중앙 집중화합니다. 컴포넌트들은 중재자 객체에 대해서만 알며 서로 직접 통신하지 않습니다.
- **중재자와 옵서버** 패턴의 차이는 종종 애매합니다. 대부분의 경우 두 패턴 중 하나를 구현할 수 있으나, 때로는 두 패턴을 동시에 적용할 수 있습니다. 이것이 어떻게 가능한지 살펴보겠습니다.

중재자의 주목적은 시스템 컴포넌트들의 집합 간의 상호 의존성을 제거하는 것입니다. 그러면 이러한 컴포넌트들은 대신 단일 중재자 객체에 의존하게 됩니다. 옵서버 패턴의 목적은 객체들 사이에 단방향 연결을 설정하는 것으로, 여기서 일부 객체는 다른 객체의 종속자 역할을 합니다.

옵서버 패턴에 의존하는 중재자 패턴의 인기 있는 구현이 있습니다. 중재자 객체는 출판사의 역할을 맡고, 컴포넌트들은 중재자의 이벤트들을 구독 및 구독 취소하는 구독자들의 역할을 맡습니다. 중재자가 이러한 방식으로 구현되면 옵서버 패턴과 매우 유사하게 보일 수 있습니다.

만약 혼란스러우시다면 중재자 패턴을 다른 방법들로 구현할 수 있음을 기억하세요. 예를 들어 모든 컴포넌트를 영구적으로 같은 중재자 객체에 연결하는 방법이 있습니다. 이 구현은 옵서버 패턴과 유사하지 않겠지만 여전히 중재자 패턴의 인스턴스일 것입니다.

이제 모든 컴포넌트가 출판사가 되어 서로 간의 동적 연결을 허용하는 프로그램을 상상해 보세요. 중앙화된 중재자 객체는 없고 분산된 옵서버들의 집합만 있을 것입니다.



메멘토 패턴

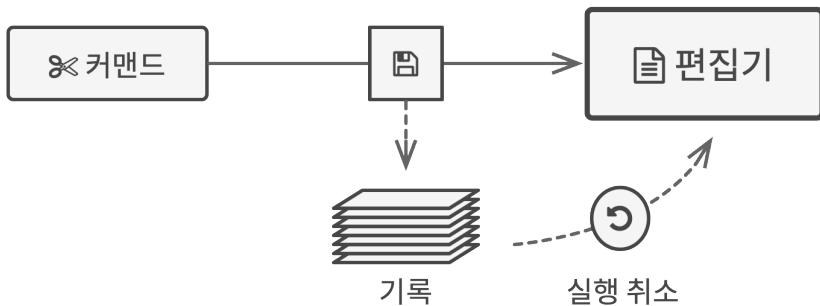
다음 이름으로도 불립니다: 스냅샷, Memento

메멘토는 객체의 구현 세부 사항을 공개하지 않으면서 해당 객체의 이전 상태를 저장하고 복원할 수 있게 해주는 행동 디자인 패턴입니다.

(:) 문제

텍스트 편집기 앱을 만들고 있다고 상상해보세요. 당신의 편집기는 간단한 텍스트 편집 외에도 텍스트의 서식 지정, 인라인 이미지들의 삽입 등을 할 수 있습니다.

어느 날 당신은 사용자들이 텍스트에 수행된 모든 작업을 실행 취소할 수 있도록 하기로 했습니다. 이 실행 취소 기능은 수년에 걸쳐 매우 보편화되었기 때문에 오늘날의 사용자들은 모든 앱에 이 기능이 있을 것이라고 가정합니다. 이 기능을 구현하기 위해 직접 접근 방식을 적용하기로 했습니다. 앱은 모든 작업을 수행하기 전에 모든 객체의 상태를 기록해 어떤 스토리지에 저장합니다. 나중에 사용자가 작업을 실행 취소하기로 하면 앱은 기록에서 가장 최신 스냅샷을 가져와 모든 객체의 상태를 복원하는 데 사용합니다.



앱은 작업을 실행하기 전에 객체들의 상태의 스냅샷을 저장하며, 이 스냅샷은 나중에 객체들을 이전 상태로 복원하는 데 사용할 수 있습니다.

상태 스냅샷들에 대해 생각해 봅시다. 상태 스냅샷은 정확히 어떻게 생성될까요? 아마도 객체의 모든 필드를 살펴본 후 해당 값들을 스토리지에 복사해야 할 것입니다. 그러나 이는 객체의 내용에 대한 액세스 제한이 상당히 완화되어 있는 경우에만 작동할 것입니다. 불행히도, 대부분의 실제 객체들은 모든 중요한 데이터를 비공개 필드에 숨깁니다.

이 문제는 일단 무시하고, 객체들이 하피족처럼 열린 관계들을 선호해 그들의 상태를 공개했다고 가정해 봅시다. 이렇게 가정하면 일단 위의 문제는 해결되어 원하는 대로 객체들의 상태에 대한 스냅샷들을 생성할 수 있습니다. 하지만 여전히 몇 가지 심각한 문제들이 남아 있습니다. 앞으로 당신이 일부 필드를 추가 또는 제거하거나, 편집기 클래스들을 리팩토링하기로 결정할지도 모르기 때문입니다. 말은 쉬워 보이지만, 그렇게 하려면 영향받은 객체들의 상태를 복사하는 역할을 맡은 클래스들을 변경해야 합니다.



객체의 비공개 상태는 어떻게 복사할까요?

그뿐만이 아닙니다. 편집기 상태의 실제 '스냅샷'들에 어떤 데이터가 포함되어 있는지 살펴봅시다. 이 안에는 최소한 실제 텍스트, 커서 좌표, 현재 스크롤 위치 등이 포함되어 있을 겁니다. 스냅샷을 만들려면 이러한 값들을 수집한 후 일종의 컨테이너에 넣어야 합니다.

아마도 당신은 이러한 컨테이너 객체들을 기록에 해당하는 어떤 리스트에 많이 저장하게 될 겁니다. 따라서 이 컨테이너들은 결국 한 클래스의 객체들이 될 것입니다. 이 클래스에는 메서드는 거의 없을 테지만, 편집기의 상태를 미러링하는 필드는 많이 있을 겁니다. 다른 객체들이 스냅샷에서 데이터를 읽고 스냅샷에 데이터를 쓸 수 있도록 하려면, 아마도 해당 스냅샷의 필드를 공개해야 할 것입니다. 그러면 편집기의 모든 (비공개 포함) 상태들이 노출될 것이고, 이제 다른 클래스들은 스냅샷 클래스에 발생하는 모든 자그마한 변경에도 영향을 받게 될 것입니다. 편집기의 모든 상태가 노출되지 않았다면 이러한 변경들은 외부 클래스에는 영향을 미치지 않은 채 비공개 필드와 메서드 안에서 변경이 발생하는 것으로 끝났을 겁니다.

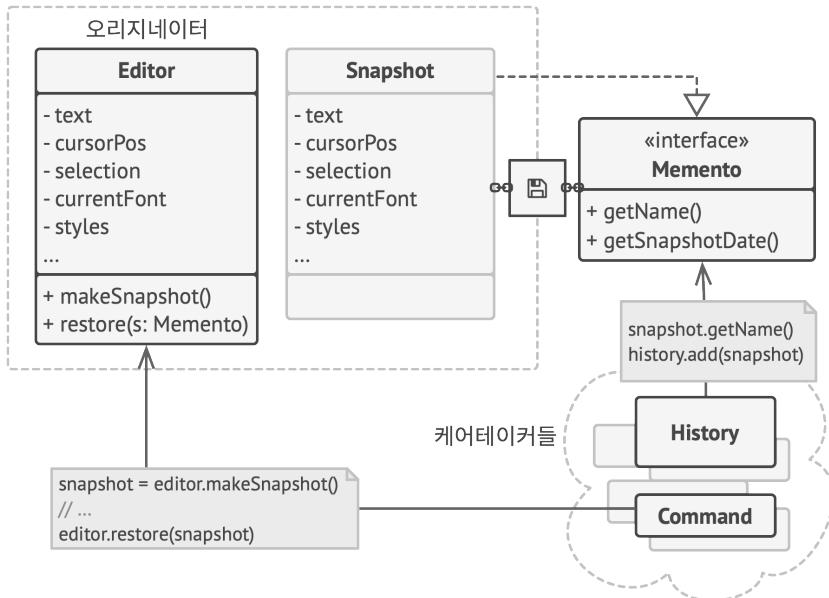
이제 교착 상태에 빠진 것 같습니다. 클래스 내부의 세부 정보를 모두 공개하면 클래스가 너무 취약해집니다. 하지만 클래스의 상태에 접근하지 못하게 하면 스냅샷을 생성할 수 없게 됩니다. 그러면 '실행 취소'는 대체 어떻게 구현해야 할까요?

呵呵 해결책

우리가 방금 경험한 모든 문제는 캡슐화의 실패로 인해 발생합니다. 일부 객체들은 원래 해야 할 일보다 더 많은 일들을 수행하려고 합니다. 예를 들어 이러한 객체들은 어떤 작업을 수행하는 데 필요한 데이터를 수집하기 위해 다른 객체들이 실제 작업을 수행하도록 놔두는 대신 그들의 비공개 공간을 침범합니다.

메멘토는 상태 스냅샷들의 생성을 해당 상태의 실제 소유자인 *originator*(オリジネイター) 객체에 위임합니다. 그러면 다른 객체들이 '외부'에서 편집기의 상태를 복사하려 시도하는 대신, 자신의 상태에 대해 완전한 접근 권한을 갖는 편집기 클래스가 자체적으로 스냅샷을 생성할 수 있습니다.

이 패턴은 메멘토라는 특수 객체에 객체 상태의 복사본을 저장하라고 제안합니다. 메멘토의 내용에는 메멘토를 생성한 객체를 제외한 다른 어떤 객체도 접근할 수 없습니다. 다른 객체들은 메멘토들과 제한된 인터페이스를 사용해 통신해야 합니다. 이러한 인터페이스는 스냅샷의 메타데이터(생성 시간, 수행한 작업의 이름, 등)를 가져올 수 있도록 할 수 있지만, 스냅샷에 포함된 원래 객체의 상태는 가져오지 못합니다.



이러한 제한 정책을 사용하면 일반적으로 케어테이커라고 하는 다른 객체들 안에 메멘토들을 저장할 수 있습니다. 케어테이커는 제한된 인터페이스를 통해서만 메멘토와 작업하기 때문에 메멘토 내부에 저장된 상태를 변경할 수 없습니다. 동시에 오리지네이터는 메멘토 내부의 모든 필드에 접근할 수 있으므로 언제든지 자신의 이전 상태를 복원할 수 있습니다.

위의 텍스트 편집기 예시의 경우, 별도의 기록 클래스를 만들어 케어테이커의 역할을 하도록 할 수 있습니다. 케어테이커 내부의 메멘토들의 스택은 편집기가 작업을 실행하려고 할 때마다 계속 늘어날 것입니다. 또 당신은 앱의 UI 내에서 이 스택을 렌더링하여

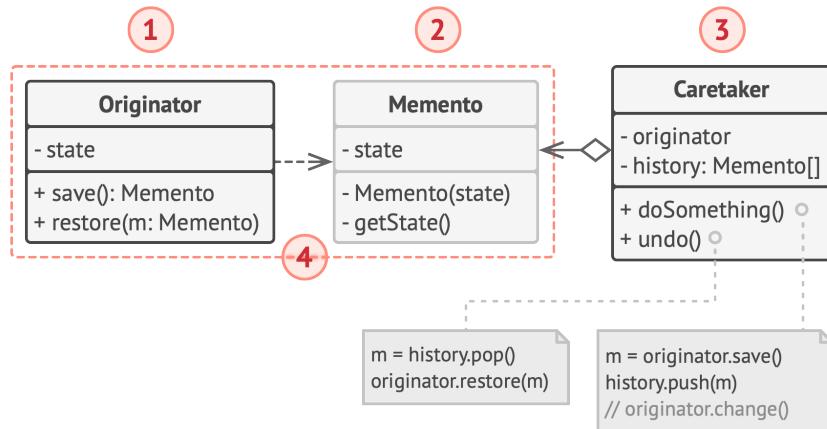
이전에 수행한 작업들의 기록을 사용자에게 표시할 수도 있습니다.

사용자가 실행 취소를 작동시키면 기록은 스택에서 가장 최근의 메멘토를 가져온 후 편집기에 다시 전달하여 롤백을 요청합니다. 편집기는 메멘토에 대한 완전한 접근 권한이 있으므로 메멘토에서 가져온 값들로 자신의 상태를 변경합니다.

구조

중첩된 클래스들에 기반한 구현

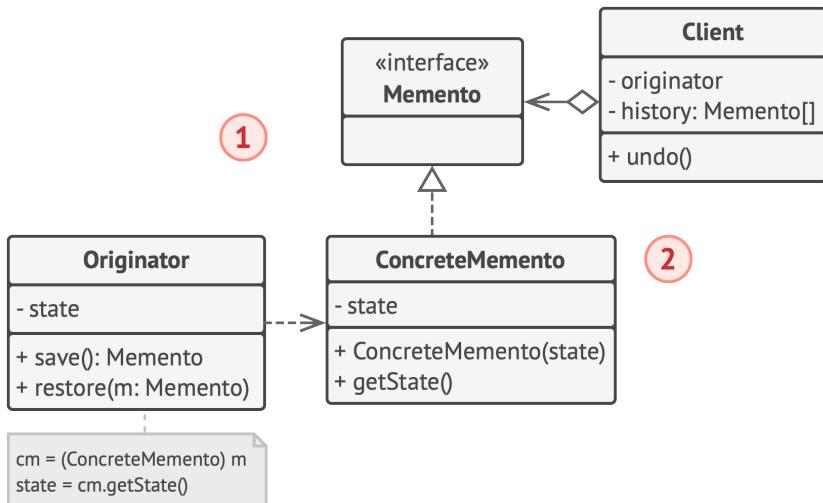
이 패턴의 고전적인 구현은 수많은 인기 프로그래밍 언어(예: C++, C# 및 자바)에서 사용할 수 있는 중첩 클래스에 대한 지원에 의존합니다.



1. **오리지네이터** 클래스는 자신의 상태에 대한 스냅샷들을 생성할 수 있으며, 필요시 스냅샷에서 자신의 상태를 복원할 수도 있습니다.
2. **메멘토**는 오리지네이터의 상태의 스냅샷 역할을 하는 값 객체입니다. 관행적으로 메멘토는 불변으로 만든 후 생성자를 통해 데이터를 한 번만 전달합니다.
3. **케어테이커**는 '언제' 그리고 '왜' 오리지네이터의 상태를 캡처해야 하는지 뿐만 아니라 상태가 복원돼야 하는 시기도 알고 있습니다.
케어테이커는 메멘토들의 스택을 저장하여 오리지네이터의 기록을 추적할 수 있습니다. 오리지네이터가 과거로 돌아가야 할 때 케어테이커는 맨 위의 메멘토를 스택에서 가져온 후 오리지네이터의 복원 메서드에 전달합니다.
4. 이 구현에서 메멘토 클래스는 오리지네이터 내부에 중첩됩니다. 이것은 오리지네이터가 메멘토의 필드들과 메서드들이 비공개로 선언된 경우에도 접근할 수 있도록 합니다. 반면에, 케어테이커는 메멘토의 필드들과 메서드들에 매우 제한된 접근 권한을 가지므로 메멘토들을 스택에 저장할 수는 있지만 그들의 상태를 변조할 수는 없습니다.

중간 인터페이스에 기반한 구현

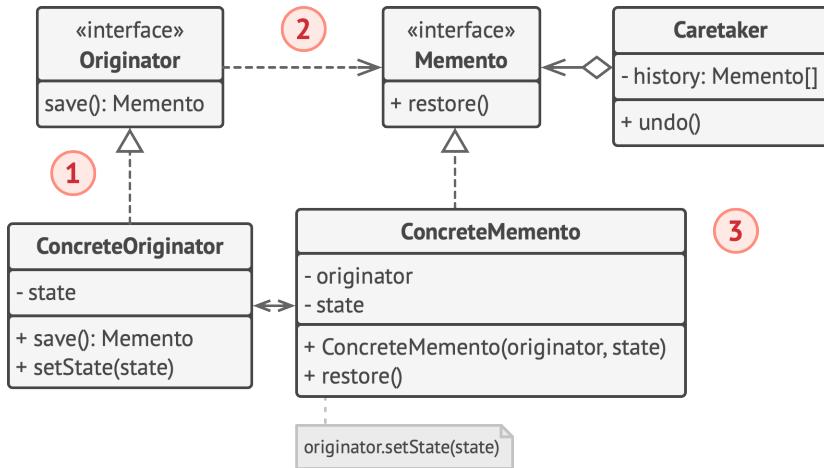
중첩 클래스들을 지원하지 않는 프로그래밍 언어(예: PHP)에 적합한 대안적 구현 방식이 있습니다.



1. 중첩 클래스들이 없는 경우, 당신은 케어테이커들이 명시적으로 선언된 중개 인터페이스를 통해서만 메멘토와 작업할 수 있는 규칙을 만들어 메멘토의 필드들에 대한 접근을 제한할 수 있습니다. 이 인터페이스는 메멘토의 메타데이터와 관련된 메서드들만 선언합니다.
2. 반면에 오리지네이터들은 메멘토 객체와 직접 작업하여 메멘토 클래스에 선언된 필드들과 메서드들에 접근할 수 있습니다. 이 접근 방식의 단점은 메멘토의 모든 구성원을 공개(public)로 선언해야 한다는 것입니다.

더 엄격한 캡슐화를 사용한 구현

또 다른 구현이 있는데, 이 구현은 당신이 다른 클래스들이 오리지네이터의 상태를 메멘토를 통해 접근할 가능성을 완전히 제거하고자 할 때 유용합니다.

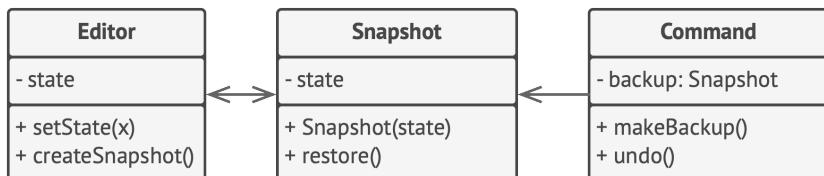


1. 이 구현 방식을 사용하면 여러 유형의 오리지네이터들과 메멘토들을 보유할 수 있습니다. 각 오리지네이터는 그에 상응하는 메멘토 클래스와 함께 작동합니다. 오리지네이터들과 메멘토들은 자신의 상태를 누구에게도 노출하지 않습니다.
2. 케어테이커들은 이제 메멘토들에 저장된 상태의 변경에 명시적인 제한을 받습니다. 또 케어테이커 클래스는 복원 메서드가 이제 메멘토 클래스에 정의되어 있으므로 오리지네이터에게서 독립됩니다.

3. 각 메멘토는 그것을 생성한 오리지네이터와 연결됩니다. 오리지네이터는 자신의 상태 값들과 함께 자신을 메멘토의 생성자에 전달합니다. 이러한 클래스 간의 긴밀한 관계 덕분에 메멘토는, 오리지네이터가 적절한 세터들을 정의했을 경우, 자신의 오리지네이터의 상태를 복원할 수 있습니다.

의사코드

이 예시에서는 메멘토를 커맨드 패턴과 함께 사용하여 복잡한 텍스트 편집기의 상태의 스냅샷들을 저장하고 필요할 때 스냅샷들로부터 이전 상태를 복원할 수 있도록 합니다.



텍스트 편집기 상태에 대한 스냅샷 저장.

커맨드 객체들은 케어테이커 역할을 합니다. 이 객체들은 커맨드들과 관련된 작업들을 실행하기 전에 편집기의 메멘토를 가져옵니다. 사용자가 가장 최근 커맨드를 실행 취소하려고 하면 편집기는 해당 커맨드에 저장된 메멘토를 사용하여 자신을 이전 상태로 되돌릴 수 있습니다.

메멘토 클래스는 공개된 필드들, 게터(getter)들 또는 세터(setter)들을 선언하지 않습니다. 따라서 어떤 객체도 자신의 내용을

변경할 수 없습니다. 메멘토들은 자신을 만든 편집기 객체에 연결됩니다. 이것은 메멘토가 데이터를 연결된 편집기 객체의 세터들을 통해 전달하여 해당 편집기의 상태를 복원할 수 있도록 합니다. 메멘토들은 특정 편집자 객체들에 연결되어 있으므로 당신은 당신의 앱이 중앙 집중식 실행 취소 스택을 사용하여 여러 독립 편집기 창을 지원하도록 할 수 있습니다.

```

1 // 오리지네이터는 시간이 지남에 따라 변경될 수 있는 어떤 중요한 데이터를
2 // 보유합니다. 또한 자신의 상태를 메멘토 내부에 저장하는 메서드와 해당 상태를
3 // 메멘토로부터 복원하는 또 다른 메서드를 정의합니다.
4 class Editor is
5     private field text, curX, curY, selectionWidth
6
7     method setText(text) is
8         this.text = text
9
10    method setCursor(x, y) is
11        this.curX = x
12        this.curY = y
13
14    method setSelectionWidth(width) is
15        this.selectionWidth = width
16
17    // 현재 상태를 메멘토 내부에 저장합니다.
18    method createSnapshot():Snapshot is
19        // 메멘토는 불변 객체입니다. 이 때문에 오리지네이터는 자신의 상태를
20        // 메멘토의 생성자 매개변수들에 전달합니다.
21        return new Snapshot(this, text, curX, curY, selectionWidth)
22
23    // 메멘토 클래스는 편집기의 이전 상태를 저장합니다.

```

```

24  class Snapshot is
25      private field editor: Editor
26      private field text, curX, curY, selectionWidth
27
28  constructor Snapshot(editor, text, curX, curY, selectionWidth) is
29      this.editor = editor
30      this.text = text
31      this.curX = x
32      this.curY = y
33      this.selectionWidth = selectionWidth
34
35 // 어느 시점에 메멘토 객체를 사용하여 편집기의 이전 상태를 복원할 수
36 // 있습니다.
37 method restore() is
38     editor.setText(text)
39     editor.setCursor(curX, curY)
40     editor.setSelectionWidth(selectionWidth)
41
42 // 커맨드 객체는 케어테이커 역할을 할 수 있습니다. 그러면 커맨드는 오리지네이터의
43 // 상태를 변경하기 직전에 메멘토를 얻습니다. 실행 취소가 요청되면 커맨드는
44 // 메멘토에서 오리지네이터의 상태를 복원합니다.
45 class Command is
46     private field backup: Snapshot
47
48 method makeBackup() is
49     backup = editor.createSnapshot()
50
51 method undo() is
52     if (backup != null)
53         backup.restore()
54     // ...

```

💡 적용

💡 **메멘토는 객체의 이전 상태를 복원할 수 있도록 객체의 상태의 스냅샷들을 생성하려는 경우에 사용하세요.**

👉 **메멘토는 비공개 필드들을 포함하여 객체의 상태의 전체 복사본들을 만들 수 있도록 하고 이 복사본들을 객체와 별도로 저장할 수 있도록 합니다.** 대부분의 개발자는 이 패턴을 '실행 취소'의 사용과 관련지어 기억하지만, 트랜잭션들을 처리할 때 (즉, 오류 발생 시 작업을 롤백해야 할 때)도 필수 불가결한 패턴입니다.

💡 **이 패턴은 또 객체의 필드들/게터들/세터들을 직접 접근하는 것이 해당 객체의 캡슐화를 위반할 때 사용하세요.**

👉 **메멘토는 객체가 스스로 자신의 상태의 스냅샷의 생성을 담당하게 합니다.** 다른 객체는 스냅샷을 읽을 수 없으므로 원래 객체의 상태 데이터는 안전합니다.

▣ 구현방법

1. 어떤 클래스가 오리지네이터의 역할을 할 것인지 결정하세요. 프로그램이 이 유형의 중심 객체를 사용하는지 아니면 여러 개의 작은 객체들을 사용하는지 아는 것이 중요합니다.

2. 메멘토 클래스를 만드세요. 하나씩 오리지네이터 클래스 내부에 선언된 필드들을 미러링하는 필드들의 집합을 선언하세요.
3. 메멘토 클래스를 변경할 수 없도록 하세요. 메멘토는 생성자를 통해 데이터를 한 번만 받아야 하며, 그 클래스에는 세터들이 없어야 합니다.
4. 사용하고 있는 프로그래밍 언어가 중첩 클래스를 지원하면 오리지네이터 내부에 메멘토를 중첩하세요. 그렇지 않은 경우, 메멘토 클래스에서 빈 인터페이스를 추출한 후 다른 모든 객체가 메멘토를 참조하는 데 사용하도록 하세요. 인터페이스에 일부 메타데이터 작업을 추가할 수 있지만 오리지네이터의 상태를 노출해서는 안 됩니다.
5. 오리지네이터 클래스에 메멘토들을 생성하는 메서드를 추가하세요. 오리지네이터는 자신의 상태를 메멘토의 생성자의 하나 또는 여러 인수들을 통해 메멘토에게 전달해야 합니다.

이 메서드의 반환 유형은 (이전 단계에서 추출했다고 가정했을 때) 이전 단계에서 추출한 인터페이스의 유형이어야 합니다. 메멘토 생성 메서드는 메멘토 클래스와 직접 작동해야 합니다.

6. 오리지네이터의 클래스에 자신의 상태를 복원하는 메서드를 추가하세요. 이 메서드는 메멘토 객체를 인수로 받아들여야 합니다. 이전 단계에서 인터페이스를 추출했다면 이 인터페이스의 유형을 매개변수의 유형으로 지정하세요. 이 경우, 들어오는

객체를 메멘토 클래스에 타입캐스팅해야 합니다. 왜냐하면 오리지네이터에게 이 객체에 대한 완전한 접근 권한이 필요하기 때문입니다.

7. 케어테이커는 커맨드 객체든, 기록이든, 아니면 완전히 다른 무언가를 나타낼 때 새로운 메멘토들을 오리지네이터로부터 언제 요청해야 하는지, 이 메멘토들을 어떻게 저장하고, 언제 특정 메멘토로부터 오리지네이터를 복원해야 하는지를 알아야 합니다.

8. 케어테이커들과 오리지네이터들 간의 연결은 메멘토 클래스로 이동시킬 수 있습니다. 이 경우, 각 메멘토는 자신을 생성한 오리지네이터와 연결되어야 합니다. 복원 메서드도 메멘토 클래스로 이동할 수 있습니다. 그러나 이 모든 것은 메멘토 클래스가 오리지네이터에 중첩되거나 오리지네이터 클래스가 메멘토 클래스의 상태를 오버라이드하기에 충분한 세터들을 제공하는 경우에만 의미가 있습니다.

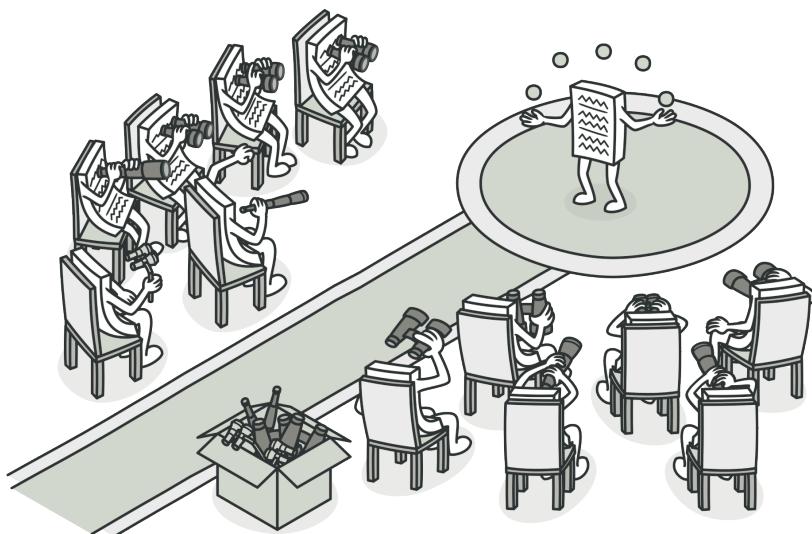
장단점

- ✓ 캡슐화를 위반하지 않고 객체의 상태의 스냅샷들을 생성할 수 있습니다.
- ✓ 당신은 케어테이커가 오리지네이터의 상태의 기록을 유지하도록 하여 오리지네이터의 코드를 단순화할 수 있습니다.

- ✖ 클라이언트들이 메멘토들을 너무 자주 생성하면 앱이 많은 RAM을 소모할 수 있습니다.
- ✖ 케어테이커들은 더 이상 쓸모없는 메멘토들을 파괴할 수 있도록 오리지네이터의 수명주기를 추적해야 합니다.
- ✖ PHP, 파이썬 및 JavaScript와 같은 대부분의 동적 프로그래밍 언어에서는 메멘토 내의 상태가 그대로 유지된다고 보장할 수 없습니다.

↔ 다른 패턴과의 관계

- 당신은 '실행 취소'를 구현할 때 커맨드와 메멘토 패턴을 함께 사용할 수 있습니다. 그러면 커맨드들은 대상 객체에 대해 다양한 작업을 수행하는 역할을 맡습니다. 반면, 메멘토들은 커맨드가 실행되기 직전에 해당 객체의 상태를 저장합니다.
- 메멘토 패턴을 반복자 패턴과 함께 사용하여 현재 순회 상태를 포착하고 필요한 경우 롤백할 수 있습니다.
- 때로는 프로토타입이 메멘토 패턴의 더 간단한 대안이 될 수 있으며, 이 패턴은 상태를 기록에 저장하려는 객체가 간단하고 외부 리소스에 대한 링크가 없거나 링크들이 있어도 이들을 재설정하기 쉬운 경우에 작동합니다.



옵서버 패턴

다음 이름으로도 불립니다: 이벤트 구독자, 경청자, Observer

옵서버 패턴은 당신이 여러 객체에 자신이 관찰 중인 객체에 발생하는 모든 이벤트에 대하여 알리는 구독 메커니즘을 정의할 수 있도록 하는 행동 디자인 패턴입니다.

(:(문제

Customer (손님) 및 **Store** (가게)라는 두 가지 유형의 객체들이 있다고 가정합니다. 손님은 곧 매장에 출시될 특정 브랜드의 제품 (예: 새 아이폰 모델)에 매우 관심이 있습니다.

손님은 매일 매장을 방문하여 제품 재고를 확인할 수 있으나, 제품이 매장에 아직 운송되는 동안 이러한 방문 대부분은 무의미합니다.



매장 방문 vs. 스팸 발송

반면 매장에서는 새로운 제품이 출시될 때마다 모든 고객에게 스팸으로 간주할 수 있는 수많은 이메일을 보낼 수 있습니다. 이 수많은 이메일은 일부 고객들을 신제품 출시 확인을 위한 잊은 매장 방문으로부터 구출해낼 수 있으나, 동시에 신제품 출시에 관심이 없는 다른 고객들을 화나게 할 것입니다.

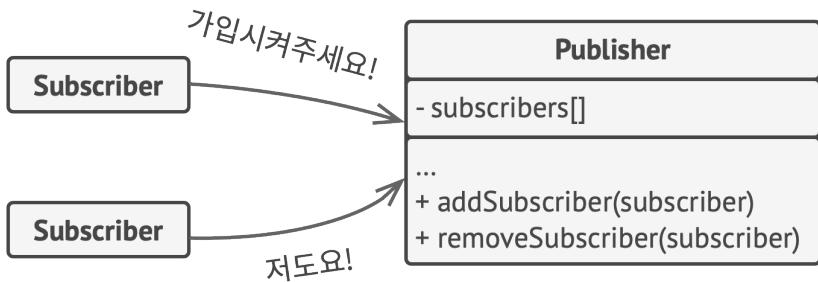
여기서 충돌이 발생합니다. 손님들이 신제품 출시 확인을 위해 시간을 낭비하든지, 매장들이 알림을 원하지 않는 고객들에게 신제품 출시를 알리며 자원을 낭비해야 합니다.

😊 해결책

시간이 지나면 변경될 수 있는 중요한 상태를 가진 객체가 있다고 가정해봅시다. 이 객체는 종종 주제(subject)라고 불립니다. 그러나 위 예시의 경우 이 객체는 자신의 상태에 대한 변경에 대해 다른 객체들에 알림을 보내는 역할도 맡을 것이니 해당 객체를 출판사라고 부르겠습니다.

읍서버 패턴은 출판사 클래스에 개별 객체들이 그 출판사로부터 오는 이벤트들의 알림들을 구독 또는 구독 취소할 수 있도록 구독 메커니즘을 추가할 것을 제안합니다. 두려워하지 마세요. 그리 복잡하지 않습니다. 실제로 이 메커니즘은 1) 구독자 객체들에 대한 참조의 리스트를 저장하기 위한 배열 필드와 2) 그 리스트에 구독자들을 추가하거나 제거할 수 있도록 하는 여러 공개된(public) 메서드들로 구성됩니다.

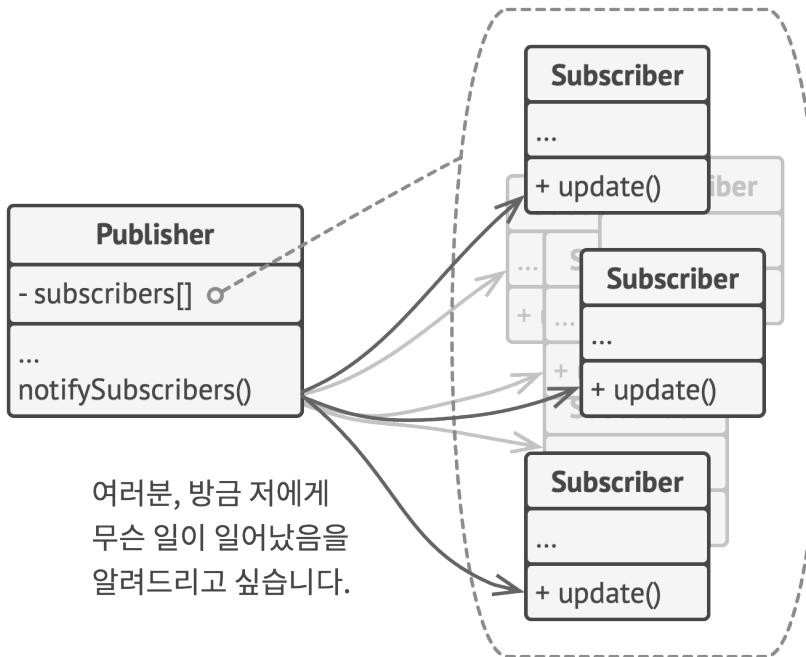
이제 출판사에 중요한 이벤트가 발생할 때마다 구독자 리스트를 참조한 후 그들의 객체들에 있는 특정 알림 메서드를 호출합니다.



구독 메커니즘을 통해 개별 객체들이 이벤트 알림들을 구독할 수 있습니다.

실제 앱에는 같은 출판사 클래스의 이벤트들을 추적하는 데 관심이 있는 수십 개의 서로 다른 구독자 클래스들이 있을 수 있습니다. 당신은 출판사를 이러한 모든 클래스에 결합하고 싶지 않을 것입니다. 게다가 당신은 당신의 출판사 클래스가 다른 사람들에 의해 사용되어야 한다면 이러한 구독자 클래스 중 일부는 미리 알지 못할 수도 있습니다.

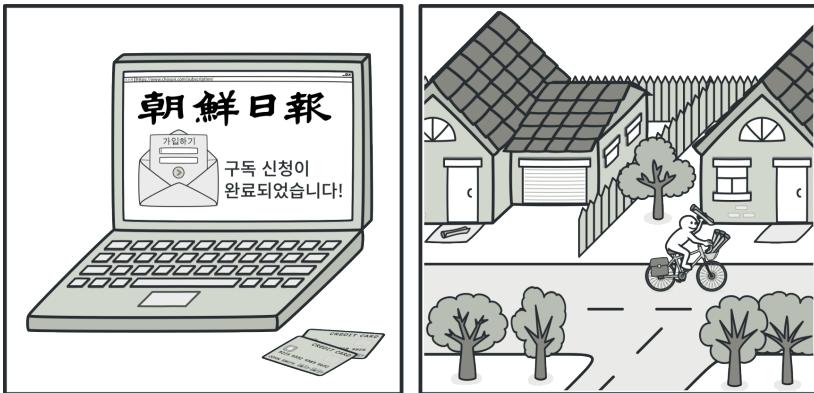
그러므로 모든 구독자가 같은 인터페이스를 구현하고 출판사가 오직 그 인터페이스를 통해서만 구독자들과 통신하는 것이 매우 중요합니다. 이 인터페이스는 출판사가 알림과 어떤 콘텍스트 데이터를 전달하는 데 사용할 수 있는 매개변수들의 집합과 알림 메서드를 선언해야 합니다.



출판사는 특정 알림 메서드를 구독자들의 객체들에서부터 호출하여 그들에게 알림을 보냅니다.

당신의 앱에 여러 유형의 출판사가 있고 이들을 구독자들과 호환되도록 하려면 당신은 더 나아가 모든 출판사가 같은 인터페이스를 따르도록 할 수 있습니다. 이 인터페이스는 몇 가지 구독 메서드들만 설명하면 됩니다. 이 인터페이스를 통해 구독자들은 출판자들의 상태들을 그들의 구상 클래스들과 결합하지 않고 관찰할 수 있습니다.

🚗 실제상황 적용

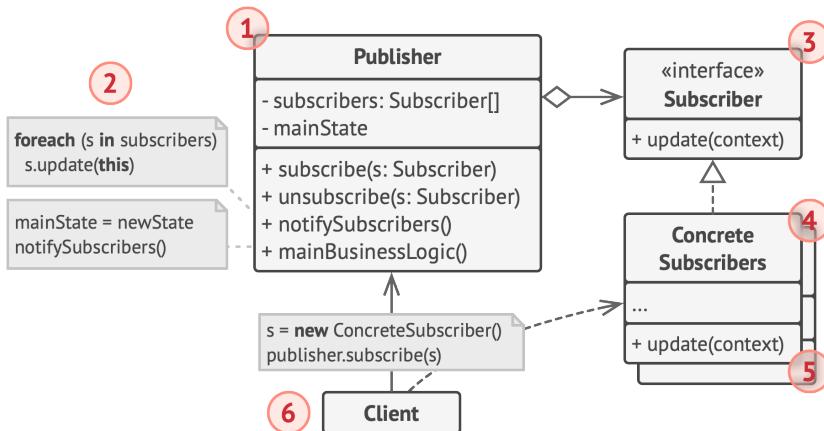


잡지 및 신문 구독.

당신이 신문이나 잡지를 구독한다면 다음 호가 있는지 확인하려 가게에 갈 필요가 없습니다. 대신 출판사가 발행 직후나 사전에 새 발행물을 구독자의 우편함으로 직접 보냅니다.

출판사는 구독자 리스트를 유지 관리하고 구독자들이 어떤 잡지에 관심 있는지 알고 있습니다. 출판사가 새로운 잡지의 발행호들을 보내는 것을 중단시키고 싶다면 구독자들은 언제든지 이 리스트를 떠날 수 있습니다.

구조

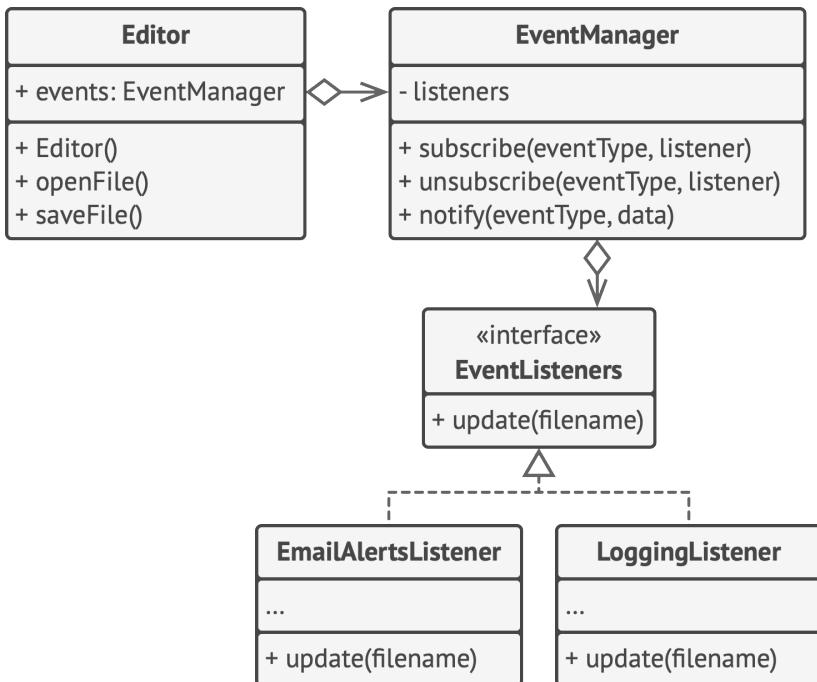


1. **출판사**는 다른 객체들에 관심 이벤트들을 발행합니다. 이러한 이벤트들은 출판사가 상태를 전환하거나 어떤 행동들을 실행할 때 발생합니다. 출판사들에는 구독 인프라가 포함되어 있으며, 이 인프라는 현재 구독자들이 리스트를 떠나고 새 구독자들이 리스트에 가입할 수 있도록 합니다.
2. 새 이벤트가 발생하면 출판사는 구독자 리스트를 살펴본 후 각 구독자 객체의 구독자 인터페이스에 선언된 알림 메서드를 호출합니다.
3. 이 **구독자** 인터페이스는 알림 인터페이스를 선언하며 대부분의 경우 단일 `update` 메서드로 구성됩니다. 이 메서드에는 출판사가 업데이트와 함께 어떤 이벤트의 세부 정보들을 전달할 수 있도록 하는 여러 매개변수가 있을 수 있습니다.

4. **구상 구독자들은** 출판사가 보낸 알림들에 대한 응답으로 몇 가지 작업을 수행합니다. 이러한 모든 클래스는 출판사가 구상 클래스들과 결합하지 않도록 같은 인터페이스를 구현해야 합니다.
5. 일반적으로 구독자들은 업데이트를 올바르게 처리하기 위해 콘텍스트 정보가 어느 정도 필요로 합니다. 그러므로 출판사들은 종종 콘텍스트 데이터를 알림 메서드의 인수들로 전달합니다. 출판사는 자신을 인수로 전달할 수 있으며, 구독자가 필요한 데이터를 직접 가져오도록 합니다.
6. **클라이언트**는 출판사 및 구독자 객체들을 별도로 생성한 후 구독자들을 출판사 업데이트에 등록합니다.

의사코드

이 예시에서 **음서버 패턴**은 텍스트 편집기 객체가 다른 서비스 객체들에 자신의 상태 변경에 대해 알릴 수 있도록 합니다.



다른 객체들에 발생하는 이벤트에 대해 객체들에 알립니다.

구독자 리스트는 동적으로 컴파일됩니다. 당신이 앱이 원하는 행동에 따라 객체들은 런타임 때 알림들을 받는 것을 시작하거나 중단할 수 있습니다.

이 구현에서 편집기 클래스는 자체적으로 구독 리스트를 유지 관리하지 않습니다. 편집기 클래스는 이 작업을 해당 작업을 전담하는 특수 도우미 객체에 위임합니다. 이 객체를 중앙 집중식 이벤트 디스패처 역할을 하도록 업그레이드하여 모든 객체가 출판사 역할을 하도록 할 수 있습니다.

앱에 새 구독자들을 추가할 때 기존 출판사 클래스들이 같은 인터페이스를 통해 모든 구독자와 작업하는 한 기존 출판사 클래스들은 변경할 필요가 없습니다.

```

1 // 기초 출판사 클래스에는 구독 관리 코드 및 알림 메서드들이 포함됩니다.
2 class EventManager is
3     private field listeners: hash map of event types and listeners
4
5     method subscribe(eventType, listener) is
6         listeners.add(eventType, listener)
7
8     method unsubscribe(eventType, listener) is
9         listeners.remove(eventType, listener)
10
11    method notify(eventType, data) is
12        foreach (listener in listeners.of(eventType)) do
13            listener.update(data)
14
15    // 구상 출판사는 일부 구독자에게 흥미로운 실제 비즈니스 논리를 포함합니다. 우리는
16    // 이 클래스를 기초 출판사로부터 파생시킬 수 있습니다. 그러나 이는 현실에서 항상
17    // 가능하지 않습니다. 왜냐하면 구상 클래스가 이미 자식 클래스일 수 있기
18    // 때문입니다. 이 경우 여기에서 했던 것처럼 합성 관계 속으로 구독 논리를 덧붙여
19    // 넣을 수 있습니다.

```

```

20  class Editor is
21      public field events: EventManager
22      private field file: File
23
24  constructor Editor() is
25      events = new EventManager()
26
27 // 비즈니스 로직의 메서드들은 구독자들에게 변경 사항에 대해 알릴 수 있습니다.
28 method openFile(path) is
29     this.file = new File(path)
30     events.notify("open", file.name)
31
32 method saveFile() is
33     file.write()
34     events.notify("save", file.name)
35
36 // ...
37
38
39 // 여기 구독자 인터페이스가 있습니다. 사용 중인 프로그래밍 언어가 함수형 타입을
40 // 지원하는 경우 전체 구독자 계층구조를 함수들의 집합으로 바꿀 수 있습니다.
41 interface EventListener is
42     method update(filename)
43
44 // 구상 구독자들은 자신이 연결된 출판사가 발행한 업데이트에 반응합니다.
45 class LoggingListener implements EventListener is
46     private field log: File
47     private field message: string
48
49 constructor LoggingListener(log_filename, message) is
50     this.log = new File(log_filename)
51     this.message = message

```

```
52
53     method update(filename) is
54         log.write(replace('%s',filename,message))
55
56 class EmailAlertsListener implements EventListener is
57     private field email: string
58     private field message: string
59
60 constructor EmailAlertsListener(email, message) is
61     this.email = email
62     this.message = message
63
64 method update(filename) is
65     system.email(email, replace('%s',filename,message))
66
67
68 // 앱은 런타임에 출판사들과 구독자들을 설정할 수 있습니다.
69 class Application is
70     method config() is
71         editor = new Editor()
72
73         logger = new LoggingListener(
74             "/path/to/log.txt",
75             "Someone has opened the file: %s")
76         editor.events.subscribe("open", logger)
77
78         emailAlerts = new EmailAlertsListener(
79             "admin@example.com",
80             "Someone has changed the file: %s")
81         editor.events.subscribe("save", emailAlerts)
```

💡 적용

💡 읍서버 패턴은 한 객체의 상태가 변경되어 다른 객체들을 변경해야 할 필요성이 생겼을 때, 그리고 실제 객체 집합들을 미리 알 수 없거나 이러한 집합들이 동적으로 변경될 때 사용하세요.

⚡ 이런 문제는 GUI 클래스와 작업할 때 자주 경험할 수 있습니다. 예를 들어 당신이 사용자 정의 버튼 클래스들을 생성했고, 이제 클라이언트들이 사용자 정의 코드를 버튼에 연결하여 사용자가 버튼을 누를 때마다 실행되도록 하고 싶다고 가정합시다.

읍서버 패턴은 구독자 인터페이스를 구현하는 모든 객체가 출판사 객체의 이벤트 알림들에 구독할 수 있도록 합니다. 당신은 버튼에 구독 메커니즘을 추가할 수 있으며, 클라이언트들이 사용자 정의 구독자 클래스들을 통해 사용자 정의 코드를 연결하도록 할 수 있습니다.

💡 이 패턴은 앱의 일부 객체들이 제한된 시간 동안 또는 특정 경우에만 다른 객체들을 관찰해야 할 때 사용하세요.

⚡ 구독 리스트는 동적이므로 구독자들은 필요할 때마다 리스트에 가입하거나 탈퇴할 수 있습니다.

구현방법

1. 당신의 앱의 비즈니스 로직을 살펴보고 두 부분으로 나누세요. 핵심 기능들은 다른 코드와 독립적이며 출판사 역할을 합니다. 나머지는 구독자 클래스들의 집합으로 바뀝니다.
2. 구독자 인터페이스를 선언하세요. 이 인터페이스는 최소한 하나의 `update` 메서드를 선언해야 합니다.
3. 출판사 인터페이스를 선언하고 구독자 객체를 구독자 리스트에 추가 및 제거하는 한 쌍의 메서드에 대해 기술하세요. 출판사들은 구독자 인터페이스를 통해서만 구독자들과 작업해야 합니다.
4. 구독 메서드들의 구현과 실제 구독 리스트를 어디에 배치할지 결정하세요. 일반적으로 모든 유형의 출판사에서 이 코드는 실질적으로 유사하므로 출판사 인터페이스에서 직접 파생된 추상 클래스에 코드를 넣는 것이 가장 적합합니다. 구상 출판사들은 이 클래스를 확장하여 해당 클래스의 구독 행동을 상속합니다.

그러나 기존 클래스 계층구조에 패턴을 적용하는 경우 합성에 기반한 접근 방식을 고려하세요. 구독 로직을 별도의 객체에 넣고 모든 실제 출판사들이 이를 사용하도록 하세요.

5. 구상 출판사 클래스들을 만드세요. 출판사 내부에서 중요한 일이 발생할 때마다 모든 구독자에게 알림을 전달해야 합니다.

6. 구상 구독자 클래스들에서 업데이트 알림 메서드들을 구현하세요. 대부분의 구독자는 이벤트에 대한 일부 컨텍스트 데이터가 필요하며, 이 데이터는 알림 메서드의 인수로 전달될 수 있습니다.

그러나 다른 옵션이 있습니다. 알림을 받으면 구독자들이 알림에서 직접 모든 데이터를 가져오도록 하는 것입니다. 이 경우 출판사는 업데이트 메서드를 통해 자신을 전달해야 합니다. 유연성이 보다 떨어지는 옵션은 생성자를 통해 출판자를 구독자에 영구적으로 연결하는 것입니다.

7. 클라이언트는 필요한 모든 구독자를 생성하고 적절한 출판사들과 등록시켜야 합니다.

△△ 장단점

- ✓ 개방/폐쇄 원칙. 출판사의 코드를 변경하지 않고도 새 구독자 클래스들을 도입할 수 있습니다. (출판사 인터페이스가 있는 경우 그 반대로 구독자의 클래스들을 변경하지 않고 새 출판사 클래스들을 도입하는 것 역시 가능합니다).
- ✓ 런타임에 객체 간의 관계들을 형성할 수 있습니다.
- ✗ 구독자들은 무작위로 알림을 받습니다.

↔ 다른 패턴과의 관계

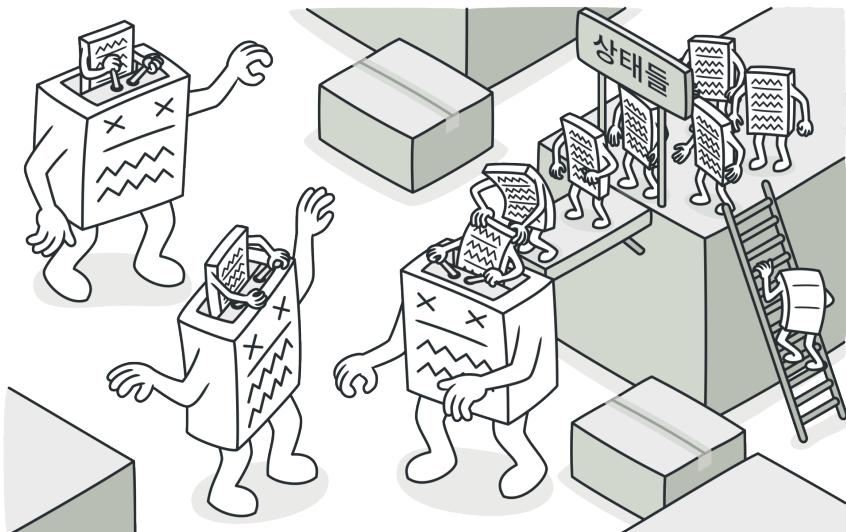
- **커맨드**, **중재자**, **옵서버** 및 **책임 연쇄** 패턴은 요청의 발신자와 수신자를 연결하는 다양한 방법을 다룹니다.
 - 책임 연쇄 패턴은 잠재적 수신자의 동적 체인을 따라 수신자 중 하나에 의해 요청이 처리될 때까지 요청을 순차적으로 전달합니다.
 - 커맨드 패턴은 발신자와 수신자 간의 단방향 연결을 설립합니다.
 - 중재자 패턴은 발신자와 수신자 간의 직접 연결을 제거하여 그들이 중재자 객체를 통해 간접적으로 통신하도록 강제합니다.
 - 옵서버 패턴은 수신자들이 요청들의 수신을 동적으로 구독 및 구독 취소할 수 있도록 합니다.
- **중재자와 옵서버** 패턴의 차이는 종종 애매합니다. 대부분의 경우 두 패턴 중 하나를 구현할 수 있으나, 때로는 두 패턴을 동시에 적용할 수 있습니다. 이것이 어떻게 가능한지 살펴보겠습니다.

중재자의 주목적은 시스템 컴포넌트들의 집합 간의 상호 의존성을 제거하는 것입니다. 그러면 이러한 컴포넌트들은 대신 단일 중재자 객체에 의존하게 됩니다. **옵서버** 패턴의 목적은 객체들 사이에 단방향 연결을 설정하는 것으로, 여기서 일부 객체는 다른 객체의 종속자 역할을 합니다.

옵서버 패턴에 의존하는 중재자 패턴의 인기 있는 구현이 있습니다. 중재자 객체는 출판사의 역할을 맡고, 컴포넌트들은 중재자의 이벤트들을 구독 및 구독 취소하는 구독자들의 역할을 맡습니다. 중재자가 이러한 방식으로 구현되면 옵서버 패턴과 매우 유사하게 보일 수 있습니다.

만약 혼란스러우시다면 중재자 패턴을 다른 방법들로 구현할 수 있음을 기억하세요. 예를 들어 모든 컴포넌트를 영구적으로 같은 중재자 객체에 연결하는 방법이 있습니다. 이 구현은 옵서버 패턴과 유사하지 않겠지만 여전히 중재자 패턴의 인스턴스일 것입니다.

이제 모든 컴포넌트가 출판사가 되어 서로 간의 동적 연결을 허용하는 프로그램을 상상해 보세요. 중앙화된 중재자 객체는 없고 분산된 옵서버들의 집합만 있을 것입니다.



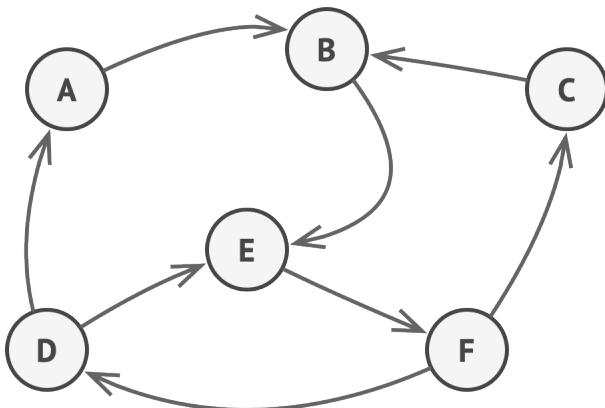
상태 패턴

다음 이름으로도 불립니다: State

상태 패턴은 객체의 내부 상태가 변경될 때 해당 객체가 그의 행동을 변경할 수 있도록 하는 행동 디자인 패턴입니다. 객체가 행동을 변경할 때 객체가 클래스를 변경한 것처럼 보일 수 있습니다.

(:) 문제

상태 패턴은 **유한 상태 기계¹** 개념과 밀접하게 관련되어 있습니다.



유한 상태 기계

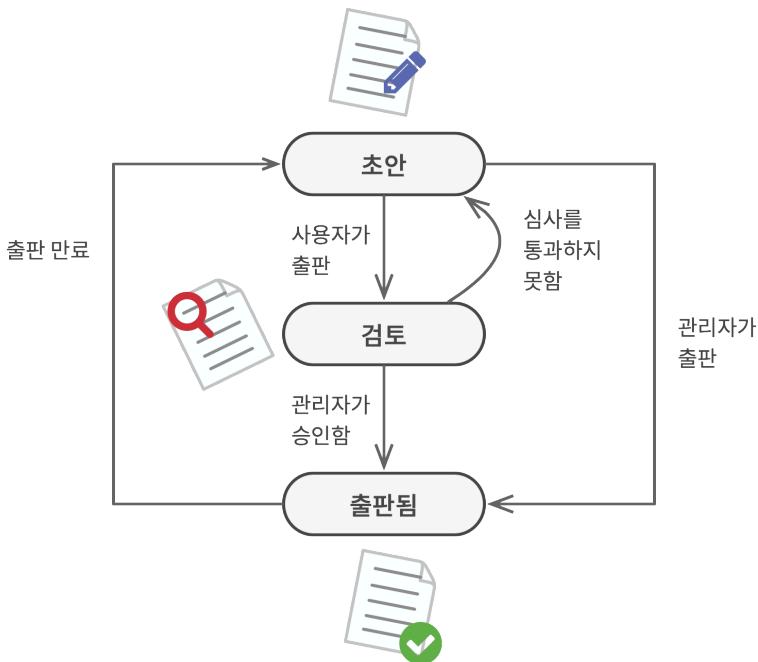
이 패턴의 주요 개념은 모든 주어진 순간에 프로그램이 속해 있을 수 있는 상태들의 수는 유한하다는 것입니다. 어떤 고유한 상태 내에서든 프로그램은 다르게 행동하며, 한 상태에서 다른 상태로 즉시 전환될 수 있습니다. 하지만 현재의 상태에 따라 프로그램은 특정 다른 상태로 전환되거나 전환되지 않을 수 있습니다. 이러한 전환 규칙들을 천이(transition)라고도 하는데, 이러한 규칙들 또한 유한하고 미리 결정되어 있습니다.

이 접근 방식을 객체들에 적용할 수도 있습니다. Document (문서) 클래스가 있다고 상상해보세요. 문서는 Draft (초안),

1. 유한 상태 기계: <https://refactoring.guru/ko/fsm>

Moderation (검토) 및 Published (출판됨)의 세 가지 상태 중 하나일 수 있습니다. 문서의 publish (출판하기) 메서드는 각 상태에서 약간씩 다르게 작동합니다.

- Draft 에서는 문서를 검토 상태로 이동합니다.
- Moderation 에서는 문서를 공개하나, 현재 사용자가 관리자인 경우에만 공개합니다.
- Published 에서는 아무 작업도 수행하지 않습니다.



문서 객체의 가능한 상태들 및 천이 (transition)들.

상태 머신들은 일반적으로 객체의 상태에 따라 적절한 행동들을 선택하는 많은 조건문 (if 또는 switch)으로 구현됩니다.

일반적으로 이 '상태'는 객체의 필드들의 값들의 집합일 뿐입니다. 당신은 유한 상태 머신에 대해 들어본 적이 없더라도 적어도 한번은 상태를 구현해봤을 것입니다. 다음 코드 구조가 익숙해 보이느냐요?

```

1  class Document is
2      field state: string
3      // ...
4  method publish() is
5      switch (state)
6          "draft":
7              state = "moderation"
8              break
9          "moderation":
10             if (currentUser.role == "admin")
11                 state = "published"
12                 break
13             "published":
14                 // Do nothing.
15                 break
16         // ...

```

조건문들에 기반한 상태 머신의 가장 큰 약점은 `Document` 클래스에 상태들과 상태에 의존하는 행동들을 추가할수록 분명해집니다. 그러면 현재 상태에 따라 메서드의 적절한 행동을 선택하는 거대한 조건문들이 대부분의 메서드들에 포함될 것입니다. 이와 같은 코드는 유지 관리하기가 매우 어렵습니다.

왜냐하면 천이 논리를 변경하려면 모든 메서드들의 상태 조건문들을 변경해야 할 수 있기 때문입니다.

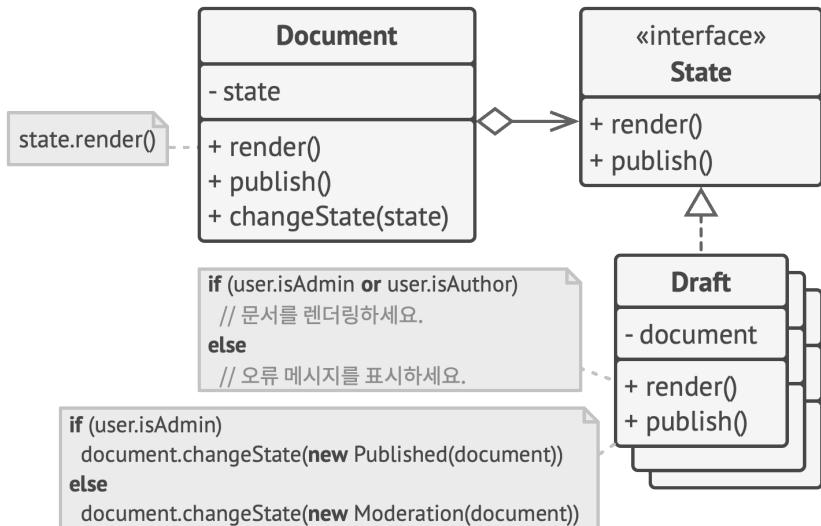
이 문제는 프로젝트가 개발되면서 더 복잡해지는 경향이 있습니다. 설계 단계에서 가능한 모든 상태와 천이를 예측하는 것은 매우 어렵습니다. 따라서, 제한된 조건문들의 집합으로 구축되어 간단명료했던 상태 머신이 시간이 지남에 따라 부풀려져 엉망이 될 수 있습니다.

◎ 해결책

상태 패턴은 객체의 모든 가능한 상태들에 대해 새 클래스들을 만들고 모든 상태별 행동들을 이러한 클래스들로 추출할 것을 제안합니다.

콘텍스트라는 원래 객체는 모든 행동을 자체적으로 구현하는 대신 현재 상태를 나타내는 상태 객체 중 하나에 대한 참조를 저장하고 모든 상태와 관련된 작업을 그 객체에 위임합니다.

콘텍스트를 다른 상태로 전환하려면 활성 상태 객체를 새 상태를 나타내는 다른 객체로 바꾸세요. 이것은 모든 상태 클래스들이 같은 인터페이스를 따르고 콘텍스트 자체가 이 인터페이스를 통해 객체들과 작동할 때만 가능합니다.



문서는 상태 객체에 작업을 위임합니다.

이 구조는 전략 패턴과 비슷해 보이지만 한 가지 중요한 차이점이 있습니다. 상태 패턴에서의 특정 상태들은 서로를 인식하고 한 상태에서 다른 상태로 천이를 시작할 수 있지만 전략들은 거의 대부분 서로에 대해 알지 못한다는 것입니다.

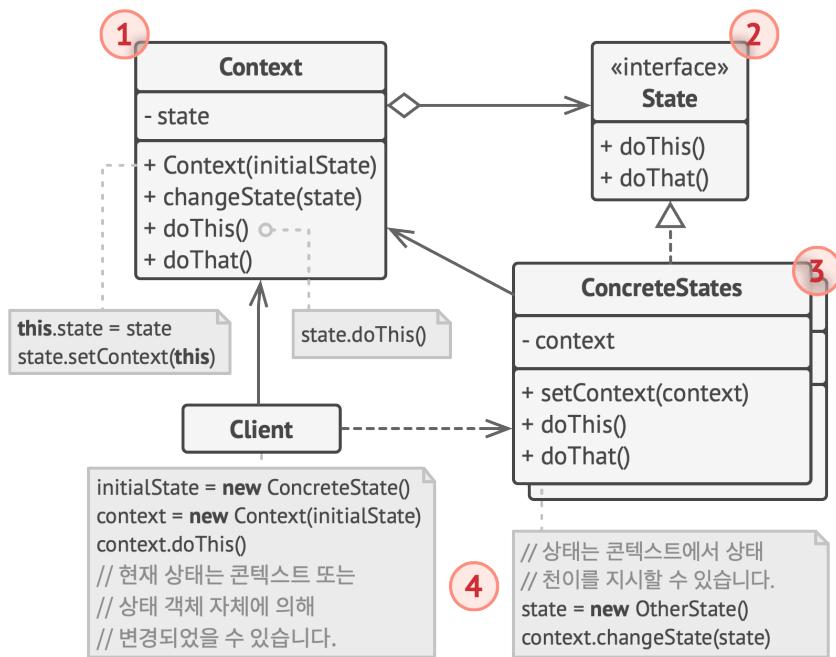
❸ 실제상황 적용

스마트폰의 버튼들과 스위치들은 장치의 현재 상태에 따라 다르게 행동합니다.

- 스마트폰이 잠금 해제된 상태에서 버튼들을 누르면 다양한 함수들이 실행됩니다.

- 스마트폰이 잠긴 상태에서 아무 버튼이나 누르면 항상 잠금 해제 화면이 나타납니다.
- 스마트폰의 충전량이 적을 때 아무 버튼이나 누르면 충전 화면이 나타납니다.

구조

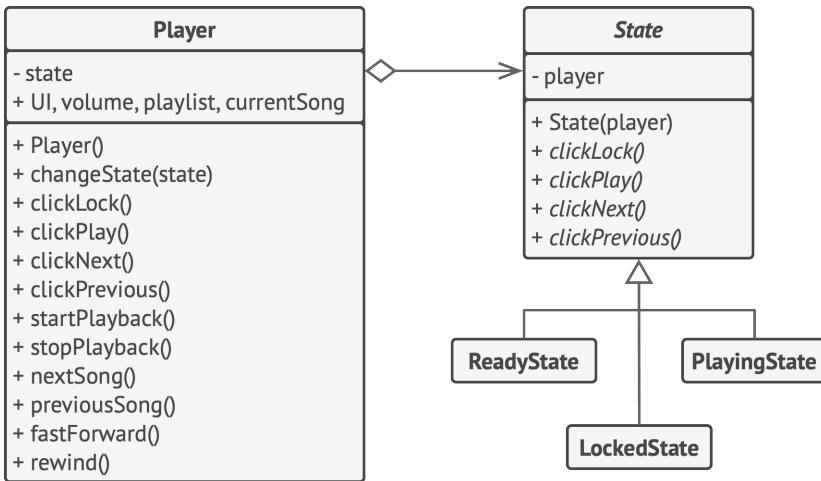


- 콘텍스트**는 구상 상태 객체 중 하나에 대한 참조를 저장하고 모든 상태별 작업을 그곳에 위임합니다. 콘텍스트는 상태 인터페이스를 통해 상태 객체와 통신하며, 새로운 상태 객체를 전달하기 위한 세터(setter)를 노출합니다.

2. **상태** 인터페이스는 상태별 메서드들을 선언합니다. 이러한 메서드들은 모든 구상 상태에서 유효해야 합니다. 왜냐하면 당신은 결코 호출될 일 없는 쓸모없는 메서드들이 일부 상태 내에 존재하는 것은 원하지 않을 것이기 때문입니다.
 3. **구상 상태들은** 상태별 메서드들에 대한 자체적인 구현을 제공합니다. 여러 상태에서 유사한 코드의 중복을 피하기 위하여 어떤 공통 행동을 캡슐화하는 중간 추상 클래스들을 제공할 수 있습니다.
- 상태 객체들은 콘텍스트 객체에 대한 역참조를 저장할 수 있습니다. 이 참조를 통해 상태는 콘텍스트 객체에서 모든 필요한 정보를 가져올 수 있고 상태 천이를 시작할 수 있습니다.
4. 콘텍스트와 구상 상태들 모두 콘텍스트의 다음 상태를 설정할 수 있으며, 콘텍스트에 연결된 상태 객체를 교체하여 실제 상태 천이를 수행할 수 있습니다.

의사코드

이 예시에서 **상태** 패턴을 사용하면 현재 재생 상태에 따라 미디어 플레이어의 같은 컨트롤들이 다르게 행동합니다.



상태 객체들을 사용하여 객체 행동을 변경하는 예시.

플레이어의 주 객체는 항상 상태 객체에 연결되며, 이 상태 객체는 플레이어를 위해 대부분 작업을 수행합니다. 일부 작업들은 플레이어의 현재 상태 객체를 다른 객체로 대체하여 플레이어가 사용자 상호 작용에 반응하는 방식을 변경합니다.

```

1 // AudioPlayer(오디오 플레이어) 클래스는 콘텍스트 역할을 합니다. 이 클래스는 또
2 // 오디오 플레이어의 현재 상태를 나타내는 상태 클래스 중 하나의 인스턴스에 대한
3 // 참조를 유지합니다.
4 class AudioPlayer is
5     field state: State
6     field UI, volume, playlist, currentSong
7
8     constructor AudioPlayer() is
9         this.state = new ReadyState(this)
10
  
```

```
11 // 콘텍스트는 사용자 입력 처리를 상태 객체에 위임합니다. 당연히 결과는
12 // 현재 활성화된 상태에 따라 달라집니다. 왜냐하면 각 상태는 입력을
13 // 다르게 처리할 수 있기 때문입니다.
14 UI = new UserInterface()
15 UI.lockButton.onClick(this.clickLock)
16 UI.playButton.onClick(this.clickPlay)
17 UI.nextButton.onClick(this.clickNext)
18 UI.prevButton.onClick(this.clickPrevious)
19
20 // 다른 객체들은 오디오 플레이어의 활성 상태를 전환할 수 있어야 합니다.
21 method changeState(state: State) is
22     this.state = state
23
24 // 사용자 인터페이스 메서드들은 실행을 활성 상태에 위임합니다.
25 method clickLock() is
26     state.clickLock()
27 method clickPlay() is
28     state.clickPlay()
29 method clickNext() is
30     state.clickNext()
31 method clickPrevious() is
32     state.clickPrevious()
33
34 // 상태는 콘텍스트에 일부 서비스 메서드들을 호출할 수 있습니다.
35 method startPlayback() is
36     ...
37 method stopPlayback() is
38     ...
39 method nextSong() is
40     ...
41 method previousSong() is
42     ...
```

```
43  method fastForward(time) is
44      // ...
45  method rewind(time) is
46      // ...
47
48
49 // 기초 상태 클래스는 모든 구상 상태들이 구현해야 하는 메서드들을 선언하고 상태와
50 // 연결된 컨텍스트 객체에 대한 역참조도 제공합니다. 상태는 역참조를 사용하여
51 // 컨텍스트를 다른 상태로 친이할 수 있습니다.
52 abstract class State is
53     protected field player: AudioPlayer
54
55     // 컨텍스트는 상태 생성자를 통해 자신을 전달합니다. 이는 필요한 경우 상태가
56     // 유용한 컨텍스트 데이터를 가져오는 데 도움이 될 수 있습니다.
57 constructor State(player) is
58     this.player = player
59
60     abstract method clickLock()
61     abstract method clickPlay()
62     abstract method clickNext()
63     abstract method clickPrevious()
64
65
66 // 구상 상태들은 컨텍스트의 상태와 연관된 다양한 행동들을 구현합니다.
67 class LockedState extends State is
68
69     // 잠긴 플레이어의 잠금을 해제하면 플레이어가 두 가지 상태 중 하나를 택할 수
70     // 있습니다.
71     method clickLock() is
72         if (player.playing)
73             player.changeState(new PlayingState(player))
74     else
```

```
75     player.changeState(new ReadyState(player))
76
77     method clickPlay() is
78         // 잠금 상태: 아무것도 하지 않는다.
79
80     method clickNext() is
81         // 잠금 상태: 아무것도 하지 않는다.
82
83     method clickPrevious() is
84         // 잠금 상태: 아무것도 하지 않는다.
85
86
87 // 콘텍스트에서 상태 천이를 실행시킬 수도 있습니다.
88 class ReadyState extends State is
89     method clickLock() is
90         player.changeState(new LockedState(player))
91
92     method clickPlay() is
93         player.startPlayback()
94         player.changeState(new PlayingState(player))
95
96     method clickNext() is
97         player.nextSong()
98
99     method clickPrevious() is
100        player.previousSong()
101
102
103 class PlayingState extends State is
104     method clickLock() is
105         player.changeState(new LockedState(player))
106
```

```

107  method clickPlay() is
108      player.stopPlayback()
109      player.changeState(new ReadyState(player))
110
111  method clickNext() is
112      if (event.doubleclick)
113          player.nextSong()
114      else
115          player.fastForward(5)
116
117  method clickPrevious() is
118      if (event.doubleclick)
119          player.previous()
120      else
121          player.rewind(5)

```

💡 적용

- ⚡ 상태 패턴은 현재 상태에 따라 다르게 행동하는 객체가 있을 때, 상태들의 수가 많을 때, 그리고 상태별 코드가 자주 변경될 때 사용하세요.
- ⚡ 이 패턴은 모든 상태별 코드를 서로 다른 클래스들의 집합으로 추출하도록 제안합니다. 그렇게 하면 새로운 상태들을 추가하거나 기존 상태들을 서로 독립적으로 변경할 수 있어서 유지 관리 비용을 절감할 수 있습니다.

☞ 이 패턴은 당신이 클래스 필드들의 현재 값들에 따라 클래스가 행동하는 방식을 변경하는 거대한 조건문들로 오염된 클래스가 있을 때 사용하세요.

↳ 상태 패턴은 당신이 이러한 조건문들의 브랜치들을 해당 상태 클래스들의 메서드들로 추출할 수 있도록 합니다. 그렇게 하는 동안 당신은 당신의 주 클래스의 상태별 코드와 관련된 임시 필드들과 도우미 메서드들을 정리할 수도 있습니다.

☞ 상태 패턴은 유사한 상태들에 중복 코드와 조건문-기반 상태 머신의 천이가 많을 때 사용하세요.

↳ 상태 패턴은 당신이 상태 클래스들의 계층구조들을 구성할 수 있도록 하며 또 공통 코드를 추상 기초 클래스들에 추출하여 중복을 줄일 수 있도록 합니다.

▣ 구현방법

1. 어떤 클래스가 콘텍스트로 작동할지 결정하세요. 이는 상태에 의존하는 코드가 이미 있는 기존 클래스일 수 있고, 상태별 코드가 여러 클래스에 분산된 경우 새로운 클래스일 수도 있습니다.
2. 상태 인터페이스를 선언하세요. 콘텍스트에 선언된 모든 메서드들을 미러링할 수 있어도 상태별 동작을 포함할 수 있는 메서드들만 목표로 설정하세요.

3. 모든 실제 상태에 대해 상태 인터페이스에서 파생된 클래스를 만드세요. 그런 다음 콘텍스트의 메서드들을 살펴보고 당신의 새로 생성된 클래스에 상태와 관련된 모든 코드를 추출하세요.

당신이 코드를 상태 클래스로 옮기는 동안 코드가 콘텍스트의 비공개 멤버들(필드와 메서드)에 의존한다는 사실을 발견할 수 있습니다. 그럴 때 몇 가지 해결 방법이 있습니다.

- 이 필드들 또는 메서드들을 공개된(public) 상태로 전환하세요.
 - 당신이 추출하는 행동을 콘텍스트의 공개된 메서드로 전환하고 상태 클래스에서 호출하세요. 이 방법은 보기 흉하지만 빠르며 나중에 언제든지 고칠 수 있습니다.
 - 사용 중인 프로그래밍 언어가 중첩 클래스들을 지원하는 경우에 한해 상태 클래스들을 콘텍스트 클래스에 중첩하세요.
4. 콘텍스트 클래스에서 상태 인터페이스 유형의 참조 필드와 필드의 값을 오버라이드할 수 있는 공개된 세터(setter)를 추가하세요.
 5. 콘텍스트의 메서드를 다시 살펴보고 빈 상태 조건문들을 상태 객체의 해당하는 메서드들에 대한 호출들로 바꾸세요.
 6. 콘텍스트의 상태를 전환하려면 상태 클래스 중 하나의 인스턴스를 만든 후 콘텍스트에 전달하세요. 당신은 이 작업을 콘텍스트 자체에서, 다양한 상태들에서, 또는 클라이언트에서 수행할 수

있습니다. 이 작업이 수행되는 곳마다 클래스는 클래스가 인스턴스화하는 구상 상태 클래스에 의존하게 됩니다.

⚠️ 장단점

- ✓ 단일 책임 원칙. 특정 상태들과 관련된 코드를 별도의 클래스들로 구성하세요.
- ✓ 개방/폐쇄 원칙. 기존 상태 클래스들 또는 콘텍스트를 변경하지 않고 새로운 상태들을 도입하세요.
- ✓ 거대한 상태 머신 조건문들을 제거하여 콘텍스트의 코드를 단순화하세요.
- ✗ 상태 머신에 몇 가지 상태만 있거나 머신이 거의 변경되지 않을 때 상태 패턴을 적용하는 것은 과도할 수 있습니다.

↔ 다른 패턴과의 관계

- **브리지, 상태, 전략** 패턴은 매우 유사한 구조로 되어 있으며, **어댑터** 패턴도 이들과 어느 정도 유사한 구조로 되어 있습니다. 위 모든 패턴은 다른 객체에 작업을 위임하는 합성을 기반으로 합니다. 하지만 이 패턴들은 모두 다른 문제들을 해결합니다. 패턴은 특정 방식으로 코드의 구조를 짜는 레시피에 불과하지 않습니다. 왜냐하면 패턴은 해결하는 문제를 다른 개발자들에게 전달할 수도 있기 때문입니다.

- **상태는 전략의 확장으로 간주할 수 있습니다.** 두 패턴 모두 합성을 기반으로 합니다. 그들은 어떤 작업을 도우미 객체들에 전달하여 콘텍스트의 행동을 바꿉니다. 전략 패턴은 이러한 객체들을 완전히 독립적으로 만들어 서로를 인식하지 못하도록 만듭니다. 그러나 상태는 구상 상태들 사이의 의존 관계들을 제한하지 않으므로 그들이 콘텍스트의 상태를 마음대로 변경할 수 있도록 합니다.



전략 패턴

다음 이름으로도 불립니다: Strategy

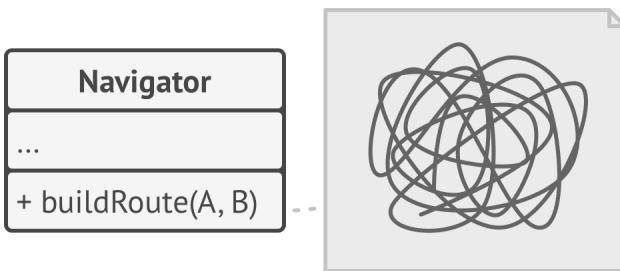
전략 패턴은 알고리즘들의 패밀리를 정의하고, 각 패밀리를 별도의 클래스에 넣은 후 그들의 객체들을 상호교환할 수 있도록 하는 행동 디자인 패턴입니다.

(:) 문제

어느 날 당신은 여행자들을 위한 내비 앱을 만들기로 했습니다. 앱의 중심 기능은 사용자들이 어느 도시에서든 빠르게 방향을 잡을 수 있도록 도와주는 아름다운 지도였습니다.

앱에서 가장 많이 요청된 기능 중 하나는 자동 경로 계획 기능이었습니다. 사용자가 주소를 입력하면 지도에 표시된 해당 목적지로 가는 가장 빠른 경로를 볼 수 있는 기능이었죠.

앱의 첫 번째 버전에서는 도로로 된 경로만을 만들 수 있었습니다. 차를 타고 여행하는 사용자들은 만족했습니다. 하지만 모든 사용자가 여가 중에 운전하는 걸 좋아하진 않았습니다. 그래서 그다음 업데이트에서는 도보 경로를 만드는 옵션을 추가했습니다. 바로 그다음에는 사람들이 경로에서 대중교통의 사용을 계획할 수 있도록 옵션을 추가했습니다.



내비게이터의 코드가 복잡해졌습니다.

하지만 그것은 시작에 불과했습니다. 나중에는 자전거를 타는 사용자들을 위한 경로를 만들 계획을 세웠습니다. 심지어 그다음에는 도시의 모든 관광 명소들을 지나는 경로를 만들 수 있는 또 다른 옵션을 추가할 계획을 세웠습니다.

사업적인 측면에서 앱은 성공했지만, 기술적인 문제들이 많은 골칫거리를 야기했습니다. 새 경로 구축 알고리즘을 추가할 때마다 내비게이터의 메인 클래스의 크기가 두 배로 늘어났으며, 어느 시점이 되자 내비 앱은 유지하기가 너무 어려워졌습니다.

간단한 버그를 수정하거나 주행거리 점수를 살짝 조정하기 위해 알고리즘 중 하나를 변경하면 전체 클래스에 영향이 미쳐 이미 작동하는 코드에서 오류가 발생할 가능성이 높아졌습니다.

또한 팀워크가 비효율적이 되었습니다. 앱 출시 직후 고용된 팀원들은 병합 충돌을 해결하는 데 너무 많은 시간을 할애해야 한다고 불평했습니다. 또 새로운 기능을 구현하려면 거대한 동일 클래스를 변경해야 했는데, 이렇게 바꾼 내용들이 다른 팀원들이 생성한 코드와 충돌하곤 했습니다.

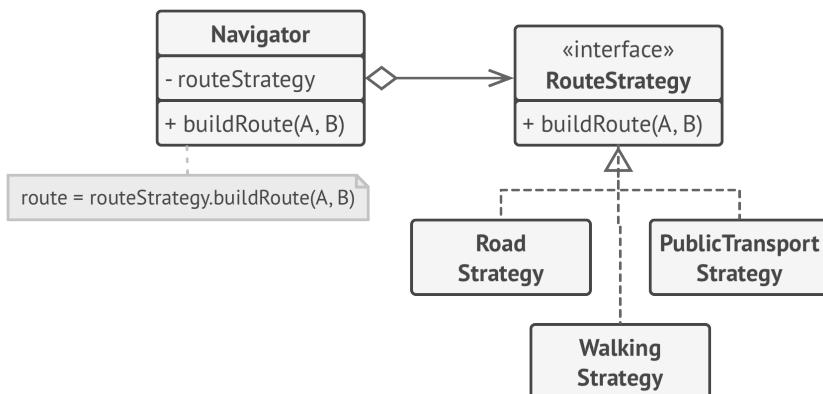
☺ 해결책

전략 패턴은 특정 작업을 다양한 방식으로 수행하는 클래스를 선택한 후 모든 알고리즘을 전략들(strategies)이라는 별도의 클래스들로 추출할 것을 제안합니다.

콘텍스트(context)라는 원래 클래스에는 전략 중 하나에 대한 참조를 저장하기 위한 필드가 있어야 합니다. 콘텍스트는 작업을 자체적으로 실행하는 대신 연결된 전략 객체에 위임합니다.

콘텍스트는 작업에 적합한 알고리즘을 선택할 책임이 없습니다. 대신 클라이언트가 원하는 전략을 콘텍스트에 전달합니다. 사실, 콘텍스트는 전략들에 대해 많이 알지 못합니다. 콘텍스트는 같은 일반 인터페이스를 통해 모든 전략과 함께 작동하며, 이 일반 인터페이스는 선택된 전략 내에 캡슐화된 알고리즘을 작동시킬 단일 메서드만 노출합니다.

이렇게 하면 콘텍스트가 구상 전략들에 의존하지 않게 되므로 콘텍스트 또는 다른 전략들의 코드를 변경하지 않고도 새 알고리즘들을 추가하거나 기존 알고리즘들을 수정할 수 있습니다.

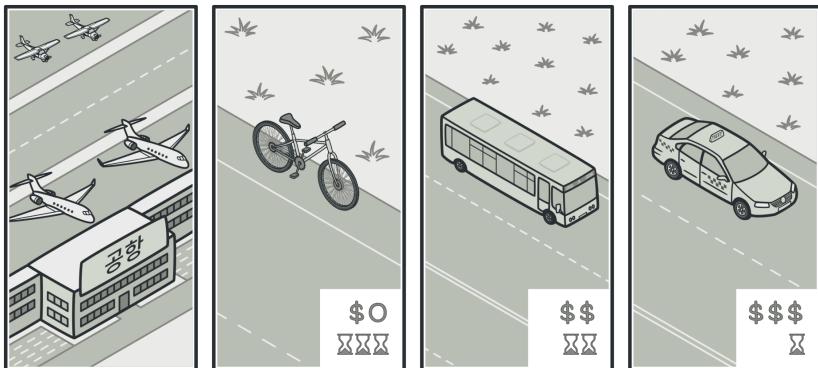


경로 계획 전략들.

당신의 내비 앱에서 각 경로 구축 알고리즘을 단일 `buildRoute` 메서드를 사용하여 자체 클래스로 추출할 수 있습니다. 이 메서드는 출발지와 목적지를 받은 후 경로의 체크포인트들의 컬렉션을 반환합니다.

같은 인수가 주어졌더라도 각 경로 구축 클래스는 다른 경로를 구축할 수 있지만 주 내비게이터 클래스는 어떤 알고리즘이 선택되었는지 별로 신경 쓰지 않습니다. 왜냐하면 주 내비게이터 클래스의 주요 작업은 지도에 체크포인트들의 집합을 렌더링하는 것이기 때문입니다. 이 클래스에는 활성 경로 구축 전략을 전환하는 메서드가 있어, 클래스의 클라이언트들이 (예: 사용자 인터페이스의 버튼들) 현재 선택된 경로 구축 행동들을 다른 행동으로 대체할 수 있습니다.

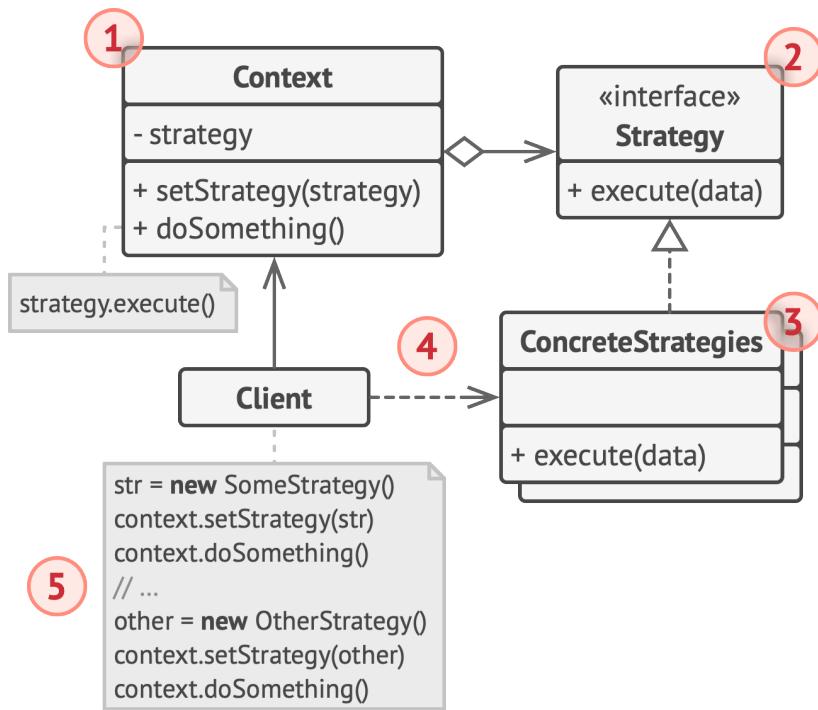
🚗 실제상황 적용



공항에 도착하기 위한 다양한 전략들.

공항에 가야 한다고 상상해 보세요. 당신은 버스를 탈 수도 있고, 택시나 자전거를 탈 수도 있습니다. 이것들이 바로 당신의 운송 전략들입니다. 예산이나 시간 제약 등을 고려하여 이러한 전략 중 하나를 선택할 수 있습니다.

구조



1. **콘텍스트**는 구상 전략 중 하나에 대한 참조를 유지하고 전략 인터페이스를 통해서만 이 객체와 통신합니다.

2. **전략** 인터페이스는 모든 구상 전략에 공통이며, 콘텍스트가 전략을 실행하는 데 사용하는 메서드를 선언합니다.

3. **구상 전략들**은 콘텍스트가 사용하는 알고리즘의 다양한 변형들을 구현합니다.

4. 콘텍스트는 알고리즘을 실행해야 할 때마다 연결된 전략 객체의 실행 메서드를 호출합니다. 콘텍스트는 알고리즘이 어떻게 실행되는지와 자신이 어떤 유형의 전략과 함께 작동하는지를 모릅니다.

5. **클라이언트**는 특정 전략 객체를 만들어 콘텍스트에 전달합니다. 콘텍스트는 클라이언트들이 런타임에 콘텍스트와 관련된 전략을 대체할 수 있도록 하는 세터(setter)를 노출합니다.

의사코드

이 예시에서의 콘텍스트는 여러 **전략들**을 사용하여 다양한 산술 연산들을 실행합니다.

```

1 // 전략 인터페이스는 어떤 알고리즘의 모든 지원 버전에 공통적인 작업을 선언합니다.
2 // 콘텍스트는 이 인터페이스를 사용하여 구상 전략들에 의해 정의된 알고리즘을
3 // 호출합니다.
4 interface Strategy is
5     method execute(a, b)
6

```

```
7 // 구상 전략들은 기초 전략 인터페이스를 따르면서 알고리즘을 구현합니다. 인터페이스는
8 // 그들이 콘텍스트에서 상호교환할 수 있게 만듭니다.
9 class ConcreteStrategyAdd implements Strategy is
10 method execute(a, b) is
11     return a + b
12
13 class ConcreteStrategySubtract implements Strategy is
14 method execute(a, b) is
15     return a - b
16
17 class ConcreteStrategyMultiply implements Strategy is
18 method execute(a, b) is
19     return a * b
20
21 // 콘텍스트는 클라이언트들이 관심을 갖는 인터페이스를 정의합니다.
22 class Context is
23     // 콘텍스트는 전략 객체 중 하나에 대한 참조를 유지합니다. 콘텍스트는 전략의
24     // 구상 클래스를 알지 못하며, 전략 인터페이스를 통해 모든 전략과 함께
25     // 작동해야 합니다.
26     private strategy: Strategy
27
28     // 일반적으로 콘텍스트는 생성자를 통해 전략을 수락하고 런타임에 전략이 전환될
29     // 수 있도록 세터도 제공합니다.
30     method setStrategy(Strategy strategy) is
31         this.strategy = strategy
32
33     // 콘텍스트는 자체적으로 여러 버전의 알고리즘을 구현하는 대신 일부 작업을 전략
34     // 객체에 위임합니다.
35     method executeStrategy(int a, int b) is
36         return strategy.execute(a, b)
37
38
```

```

39 // 클라이언트 코드는 구상 전략을 선택하고 컨텍스트에 전달합니다. 클라이언트는 올바른
40 // 선택을 하기 위해 전략 간의 차이점을 알고 있어야 합니다.
41 class ExampleApplication is
42     method main() is
43         Create context object.
44
45         Read first number.
46         Read last number.
47         Read the desired action from user input.
48
49         if (action == addition) then
50             context.setStrategy(new ConcreteStrategyAdd())
51
52         if (action == subtraction) then
53             context.setStrategy(new ConcreteStrategySubtract())
54
55         if (action == multiplication) then
56             context.setStrategy(new ConcreteStrategyMultiply())
57
58         result = context.executeStrategy(First number, Second number)
59
60         Print result.

```

💡 적용

 전략 패턴은 객체 내에서 한 알고리즘의 다양한 변형들을 사용하고 싶을 때, 그리고 런타임 중에 한 알고리즘에서 다른 알고리즘으로 전환하고 싶을 때 사용하세요.

- ⚡ 또 전략 패턴은 객체의 행동들을 특정 하위 행동들을 다양한 방식으로 수행할 수 있는 다른 하위 객체들과 연관시켜 객체의 행동들을 런타임에 간접적으로 변경할 수 있게 해줍니다.
- ⚡ 전략 패턴은 일부 행동을 실행하는 방식에서만 차이가 있는 유사한 클래스들이 많은 경우에 사용하세요.
- ⚡ 전략 패턴은 다양한 행동들을 별도의 클래스 계층구조로 추출하고 원래 클래스들을 하나로 결합하여 중복 코드를 줄일 수 있게 해줍니다.
- ⚡ 전략 패턴을 사용하여 클래스의 비즈니스 로직을 해당 로직의 콘텍스트에서 그리 중요하지 않을지도 모르는 알고리즘들의 구현 세부 사항들로부터 고립하세요.
- ⚡ 전략 패턴은 코드의 나머지 부분에서 해당 코드, 내부 데이터, 그리고 다양한 알고리즘들의 의존 관계들을 고립시킬 수 있습니다. 다양한 클라이언트들이 알고리즘들을 실행하고 런타임에 전환하기 위한 간단한 인터페이스를 얻습니다.
- ⚡ 이 패턴은 같은 알고리즘의 다른 변형들 사이를 전환하는 거대한 조건문이 당신의 클래스에 있는 경우에 사용하세요.
- ⚡ 전략 패턴을 사용하면 모든 알고리즘을 같은 인터페이스를 구현하는 별도의 클래스들로 추출하여 이러한 조건문을 제거할 수

있습니다. 원래 객체는 알고리즘의 모든 변형들을 구현하는 대신 이러한 객체들 중 하나에 실행을 위임합니다.

▣ 구현방법

1. 콘텍스트 클래스에서 자주 변경되는 알고리즘을 식별하세요. 런타임에 같은 알고리즘의 변형을 선택한 후 실행하는 거대한 조건문일 수도 있습니다.
2. 알고리즘의 모든 변형에 공통인 전략 인터페이스를 선언하세요.
3. 하나씩 모든 알고리즘을 자체 클래스들로 추출하세요. 그들은 모두 전략 인터페이스를 구현해야 합니다.
4. 콘텍스트 클래스에서 전략 객체에 대한 참조를 저장하기 위한 필드를 추가한 후, 해당 필드의 값을 대체하기 위한 세터를 제공하세요. 콘텍스트는 전략 인터페이스를 통해서만 전략 객체와 작동해야 합니다. 콘텍스트는 인터페이스를 정의할 수 있으며, 이 인터페이스는 전략이 콘텍스트의 데이터에 접근할 수 있도록 합니다.
5. 콘텍스트의 클라이언트들은 콘텍스트를 적절한 전략과 연관시켜야 합니다. 이러한 전략은 클라이언트들이 기대하는 콘텍스트가 주 작업을 수행하는 방식과 일치해야 합니다.

장단점

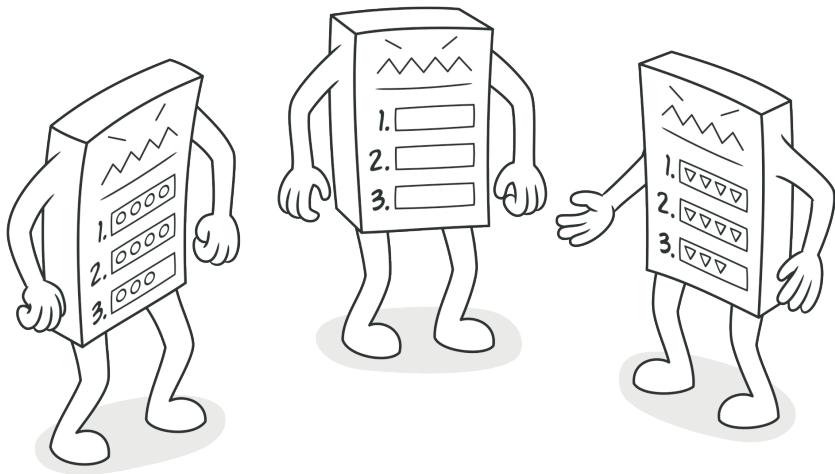
- ✓ 런타임에 한 객체 내부에서 사용되는 알고리즘들을 교환할 수 있습니다.
- ✓ 알고리즘을 사용하는 코드에서 알고리즘의 구현 세부 정보들을 고립할 수 있습니다.
- ✓ 상속을 합성으로 대체할 수 있습니다.
- ✓ 개방/폐쇄 원칙. 콘텍스트를 변경하지 않고도 새로운 전략들을 도입할 수 있습니다.

- ✗ 알고리즘이 몇 개밖에 되지 않고 거의 변하지 않는다면, 패턴과 함께 사용되는 새로운 클래스들과 인터페이스들로 프로그램을 지나치게 복잡하게 만들 이유가 없습니다.
- ✗ 클라이언트들은 적절한 전략을 선택할 수 있도록 전략 간의 차이점을 알고 있어야 합니다.
- ✗ 현대의 많은 프로그래밍 언어에는 익명 함수들의 집합 내에서 알고리즘의 다양한 버전들을 구현할 수 있는 함수형 지원이 있으며, 클래스들과 인터페이스들을 추가하여 코드의 부피를 늘리지 않으면서도 전략 객체를 사용했을 때와 똑같이 이러한 함수들을 사용할 수 있습니다.

↔ 다른 패턴과의 관계

- **브리지**, **상태**, **전략** 패턴은 매우 유사한 구조로 되어 있으며, **어댑터** 패턴도 이들과 어느 정도 유사한 구조로 되어 있습니다. 위 모든 패턴은 다른 객체에 작업을 위임하는 합성을 기반으로 합니다. 하지만 이 패턴들은 모두 다른 문제들을 해결합니다. 패턴은 특정 방식으로 코드의 구조를 짜는 레시피에 불과하지 않습니다. 왜냐하면 패턴은 해결하는 문제를 다른 개발자들에게 전달할 수도 있기 때문입니다.
- **커맨드**와 **전략** 패턴은 비슷해 보일 수 있습니다. 왜냐하면 둘 다 어떤 작업으로 객체를 매개변수화하는 데 사용할 수 있기 때문입니다. 그러나 이 둘의 의도는 매우 다릅니다.
 - 당신은 커맨드^{•••}를 사용하여 모든 작업을 객체로 변환할 수 있습니다. 작업의 매개변수들은 해당 객체의 필드들이 됩니다. 이 변환은 작업의 실행을 연기하고, 해당 작업을 대기열에 넣고, 커맨드들의 기록을 저장한 후 해당 커맨드들을 원격 서비스에 보내는 등의 작업을 가능하게 합니다.
 - 반면에 전략^{•••} 패턴은 일반적으로 같은 작업을 수행하는 다양한 방법을 설명하므로 단일 콘텍스트 클래스 내에서 이러한 알고리즘들을 교환할 수 있도록 합니다.
- **데코레이터**는 객체의 피부를 변경할 수 있고 **전략** 패턴은 객체의 내장을 변경할 수 있다고 비유할 수 있습니다.

- **템플릿 메서드는** 상속을 기반으로 합니다. 이 메서드는 자식 클래스들에서 알고리즘의 부분들을 확장하여 변경할 수 있도록 합니다. **전략 패턴은** 합성을 기반으로 합니다: 당신은 객체 행동의 일부분들을 이러한 행동에 해당하는 다양한 전략들을 제공하여 변경할 수 있습니다. **템플릿 메서드는** 클래스 수준에서 작동하므로 정적입니다. **전략 패턴은** 객체 수준에서 작동하므로 런타임에 행동들을 전환할 수 있도록 합니다.
- **상태는 전략의 확장으로** 간주할 수 있습니다. 두 패턴 모두 합성을 기반으로 합니다. 그들은 어떤 작업을 도우미 객체들에 전달하여 콘텍스트의 행동을 바꿉니다. **전략 패턴은** 이러한 객체들을 완전히 독립적으로 만들어 서로를 인식하지 못하도록 만듭니다. 그러나 **상태는** 구상 상태들 사이의 의존 관계들을 제한하지 않으므로 그들이 콘텍스트의 상태를 마음대로 변경할 수 있도록 합니다.



템플릿 메서드 패턴

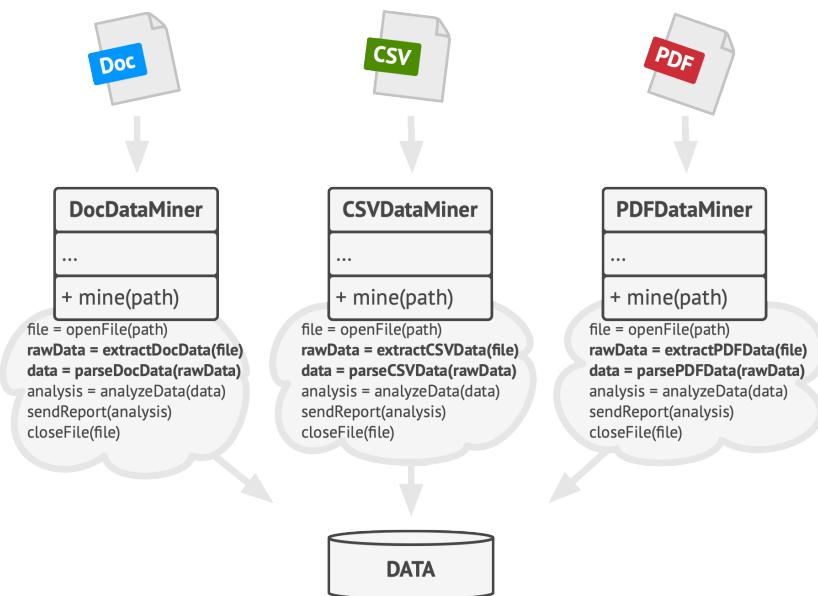
다음 이름으로도 불립니다: Template Method

템플릿 메서드는 부모 클래스에서 알고리즘의 골격을 정의하지만, 해당 알고리즘의 구조를 변경하지 않고 자식 클래스들이 알고리즘의 특정 단계들을 오버라이드(재정의)할 수 있도록 하는 행동 디자인 패턴입니다.

:(문제

회사 문서들을 분석하는 데이터 마이닝 앱을 만들고 있다고 가정해 봅시다. 사용자들은 앱에 다양한 형식(PDF, DOC, CSV)의 문서들을 제공하고 앱은 이러한 문서들에서 일관된 형식으로 의미 있는 데이터를 추출하려고 시도합니다.

앱의 첫 번째 버전은 DOC 파일과만 작동할 수 있었고, 다음 버전에서는 CSV 파일을 지원할 수 있었습니다. 한 달 후, 당신은 앱이 PDF 파일에서 데이터를 추출하도록 '가르쳤습니다'.



데이터 마이닝 클래스들에는 중복 코드가 많이 포함되어 있습니다.

어느 날 당신은 세 클래스 모두에 유사한 코드가 많다는 것을 알게 되었습니다. 다양한 데이터 형식들을 처리하는 코드는 클래스마다 완전히 다르지만 데이터 처리 및 분석을 위한 코드는 거의 같습니다. 알고리즘 구조는 그대로 두되, 코드 중복은 제거하는 게 좋지 않을까요?

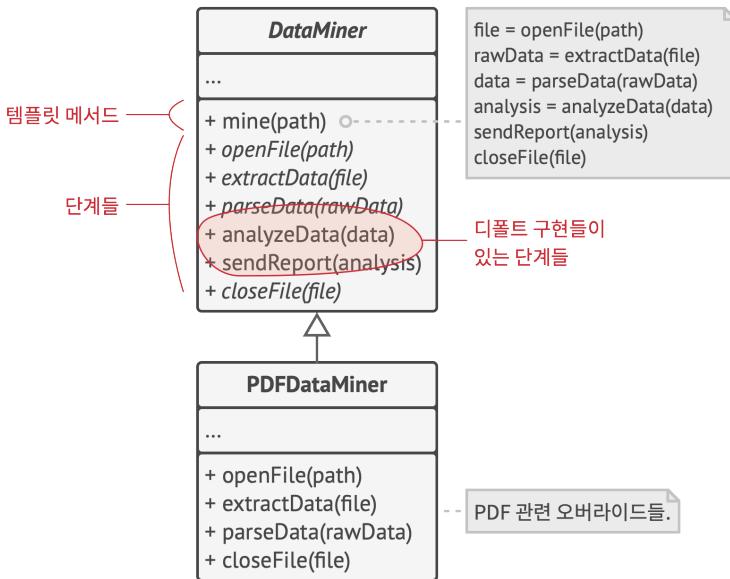
이 클래스들을 사용하는 클라이언트 코드와 관련된 또 다른 문제가 있었습니다. 이 코드에는 작업을 처리하고 있는 클래스에 따라 적절한 행동들을 선택하는 조건문이 많이 있었습니다. 세 처리 클래스에 전부 공통 인터페이스나 공통 기초 클래스가 있었다면, 클라이언트 코드에서 조건문들을 제거하고 처리 객체에 메서드를 호출할 때 다형성을 사용할 수 있었을 겁니다.

呵呵 해결책

템플릿 메서드 패턴은 알고리즘을 일련의 단계들로 나누고, 이러한 단계들을 메서드들로 변환한 뒤, 단일 템플릿 메서드 내부에 이러한 메서드들에 대한 일련의 호출들을 넣으라고 제안합니다. 이러한 단계들은 *abstract* (추상)이거나 일부 디폴트 (기본값) 구현을 가질 것입니다. 알고리즘을 사용하기 위해 클라이언트는 자신의 자식 클래스를 제공해야 하고, 모든 추상 단계를 구현해야 하며, 필요하다면 (템플릿 메서드를 제외한) 선택적 단계 중 일부를 오버라이드(재정의)해야 합니다.

이것이 당신의 데이터 마이닝 앱에서 어떻게 작동하는지 봅시다. 세 가지 파싱 알고리즘들 모두를 위한 기초 클래스를 만들 수

있습니다. 이 기초 클래스는 다양한 문서 처리 단계들에 대한 일련의 호출들로 구성된 템플릿 메서드를 정의합니다.



템플릿 메서드는 알고리즘을 단계로 나누어 자식 클래스들이 위에 언급된 실제 메서드를 제외한 이 단계들을 오버라이드할 수 있도록 합니다.

처음에는 모든 단계를 `abstract`로 선언하여 자식 클래스들이 이러한 메서드들에 대한 자체 구현을 제공하도록 강제할 수 있습니다. 당신의 앱의 경우, 자식 클래스들은 이미 필요한 모든 구현들을 가지고 있으므로, 우리가 해야 할 유일한 일은 메서드들의 시그니처들을 부모 클래스의 메서드들과 일치하도록 조정하는 것입니다.

이제 중복 코드를 제거하기 위해 무엇을 할 수 있는지 봅시다. 파일 열기/닫기 및 데이터 추출/파싱을 위한 코드는 데이터 형식들에

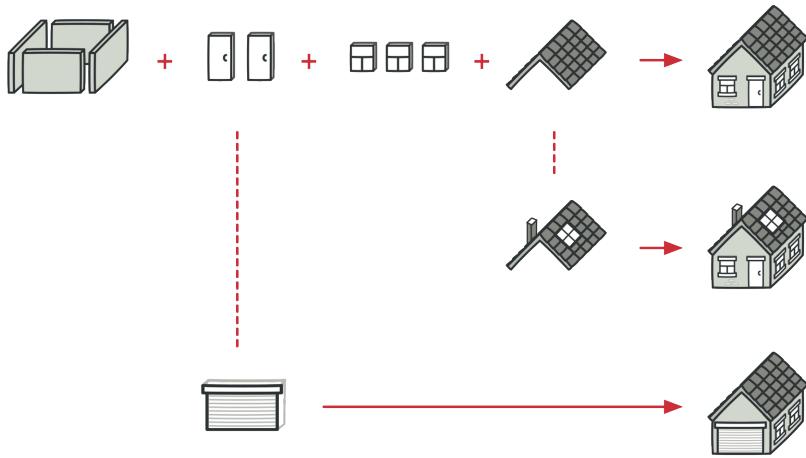
따라 다르므로 해당 메서드들을 건드릴 의미가 없습니다. 그러나 미가공 데이터 분석 및 보고서 작성과 같은 다른 단계들의 구현은 매우 유사하므로 기초 클래스로 끌어올릴 수 있습니다. 그러면 자식 클래스들은 기초 클래스에서 이 코드를 공유할 수 있습니다.

보시다시피 두 가지 유형의 단계들이 있습니다:

- 모든 자식 클래스는 추상 단계들을 구현해야 합니다.
- 선택적 단계들에는 이미 어떤 디폴트(기본값) 구현이 있지만, 필요한 경우 이를 무시하고 오버라이드(재정의) 할 수 있습니다.

흙이라는 또 다른 유형의 단계가 있습니다. 흙은 물체가 비어 있는 선택적 단계입니다. 템플릿 메서드는 흙이 오버라이드 되지 않아도 작동합니다. 일반적으로 흙들은 알고리즘의 중요한 단계들의 전 또는 후에 배치되어 자식 클래스들에 알고리즘에 대한 추가 확장 지점들을 제공합니다.

☞ 실제상황 적용

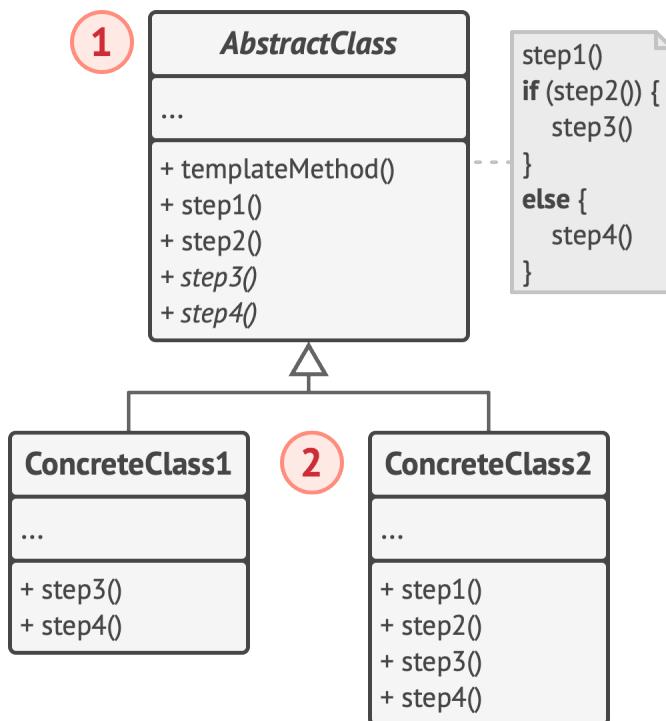


일반적인 건축 계획은 클라이언트의 니즈에 더 잘 부합하도록 약간 변경될 수 있습니다.

템플릿 메서드 접근 방식은 대량 주택 건설에 사용할 수 있습니다. 표준 주택 건설을 위한 건축 계획에는 잠재적 주택 소유자가 결과 주택의 일부 세부 사항들을 조정할 수 있도록 하는 여러 확장 지점들이 포함될 수 있습니다.

완성된 집이 다른 집들과 조금씩 다르도록 각 건축 단계(예: 기초 쌓기, 골조공사, 벽 쌓기, 수도 및 전기 배선 설치 등)는 약간씩 변경될 수 있습니다.

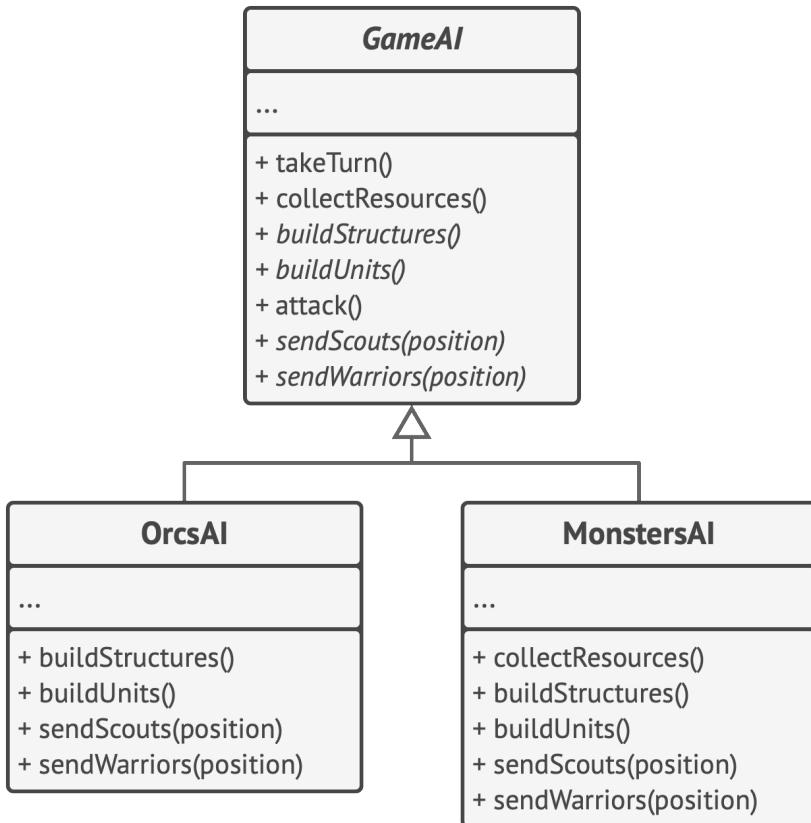
템플릿 구조



- 추상 클래스는 알고리즘의 단계들의 역할을 하는 메서드들을 선언하며, 이러한 메서드를 특정 순서로 호출하는 실제 템플릿 메서드도 선언합니다. 단계들은 `abstract`로 선언되거나 일부 디폴트 구현을 갖습니다.**
- 구상 클래스들은 모든 단계들을 오버라이드할 수 있지만 템플릿 메서드 자체는 오버라이드 할 수 없습니다.**

의사코드

이 예시에서 **템플릿 메서드** 패턴은 간단한 전략 비디오 게임의 인공 지능의 다양한 브랜치들에 대한 '골격'을 제공합니다.



간단한 비디오 게임의 인공지능 클래스들.

게임의 모든 종족은 거의 같은 유형의 유닛들과 건물들을 가지고 있습니다. 따라서 다양한 종족에 대해 같은 인공지능 구조를 재사용하면서 일부 세부 사항들은 오버라이드할 수 있습니다. 이

접근 방식을 사용하면 오크들의 인공지능을 오버라이드하여 그들을 더 공격적으로 만들고, 같은 방식으로 인간들을 방어 지향적으로 만들고 몬스터들은 아무것도 건설할 수 없도록 만들 수 있습니다. 게임에 새 종족을 추가하려면 새 인공지능 자식 클래스를 만들고 기초 인공지능 클래스에 선언된 디폴트 메서드들을 오버라이드해야 합니다.

```

1 // 추상 클래스는 템플릿 메서드를 정의합니다. 이 메서드는 일반적으로 원시 작업을
2 // 추상화하기 위해 호출로 구성된 어떤 알고리즘의 골격을 포함합니다. 구상 자식
3 // 클래스들은 이러한 작업을 구현하지만 템플릿 메서드 자체는 그대로 둡니다.
4 class GameAI is
5   // 템플릿 메서드는 알고리즘의 골격을 정의합니다.
6   method turn() is
7     collectResources()
8     buildStructures()
9     buildUnits()
10    attack()
11
12  // 일부 단계들은 기초 클래스에서 바로 구현될 수 있습니다.
13  method collectResources() is
14    foreach (s in this.builtStructures) do
15      s.collect()
16
17  // 그리고 그중 일부는 추상으로 정의될 수 있습니다.
18  abstract method buildStructures()
19  abstract method buildUnits()
20
21  // 한 클래스에는 여러 템플릿 메서드가 있을 수 있습니다.
22  method attack() is
23    enemy = closestEnemy()

```

```
24     if (enemy == null)
25         sendScouts(map.center)
26     else
27         sendWarriors(enemy.position)
28
29     abstract method sendScouts(position)
30     abstract method sendWarriors(position)
31
32 // 구상 클래스들은 기초 클래스의 모든 추상 작업을 구현해야 합니다. 하지만 템플릿
33 // 메서드 자체를 오버라이드해서는 안 됩니다.
34 class OrcsAI extends GameAI {
35     method buildStructures() {
36         if (there are some resources) {
37             // 농장들, 막사들, 그리고 요새들을 차례로 건설하세요.
38
39         method buildUnits() {
40             if (there are plenty of resources) {
41                 if (there are no scouts)
42                     // 잡역인을 생성한 후 정찰병 그룹에 추가하세요.
43                 else
44                     // 하급 병사를 생성한 후 전사 그룹에 추가하세요.
45
46         // ...
47
48     method sendScouts(position) {
49         if (scouts.length > 0) {
50             // 정찰병들을 위치로 보내세요.
51
52     method sendWarriors(position) {
53         if (warriors.length > 5) {
54             // 전사들을 위치로 보내세요.
```

```

56 // 자식 클래스들은 디폴트 구현을 가진 일부 작업을 오버라이드할 수 있습니다.
57 class MonstersAI extends GameAI {
58     method collectResources() {
59         // 몬스터들은 자원을 모으지 않습니다.
60     }
61     method buildStructures() {
62         // 몬스터들은 건물을 짓지 않습니다.
63     }
64     method buildUnits() {
65         // 몬스터들은 유닛들을 생성하지 않습니다.

```

💡 적용

💡 템플릿 메서드 패턴은 클라이언트들이 알고리즘의 특정 단계들만 확장할 수 있도록 하고 싶을 때, 그러나 전체 알고리즘이나 알고리즘 구조는 확장하지 못하도록 하려고 할 때 사용하세요.

💡 템플릿 메서드는 모듈리식 알고리즘을 일련의 개별 단계들로 전환할 수 있도록 합니다. 이 알고리즘은 부모 클래스에서 정의된 구조를 그대로 유지하면서 자식 클래스들에 의해 쉽게 확장될 수 있습니다.

💡 이 패턴은 약간의 차이가 있지만 거의 같은 알고리즘들을 포함하는 여러 클래스가 있는 경우에 사용하세요. 결과적으로 알고리즘이 변경되면 모든 클래스를 수정해야 할 수도 있습니다.

▶ 이러한 알고리즘을 템플릿 메서드로 전환하면 유사한 구현들이 있는 단계들을 부모 클래스로 끌어올릴 수 있으며, 그로 인해 코드 중복을 제거할 수 있습니다. 자식 클래스 중 서로 코드가 다른 부분들은 자식 클래스들에 남겨놓을 수 있습니다.

▣ 구현방법

1. 대상 알고리즘을 분석하여 여러 단계로 나눌 수 있는지 확인하세요. 어떤 단계들이 모든 자식 클래스에 공통인지 또 어떤 단계들이 항상 고유한지를 고려하세요.
2. 추상 기초 클래스를 만들고 알고리즘의 단계들을 표현하는 템플릿 메서드와 추상 메서드들의 집합을 선언하세요. 해당하는 단계들을 실행하여 템플릿 메서드에서 알고리즘의 구조의 윤곽을 잡으세요. 템플릿 메서드를 `final`로 만들어 자식 클래스들이 메서드를 오버라이드하지 못하도록 하는 것을 고려하세요.
3. 모든 단계가 추상적이어도 괜찮습니다. 그러나 일부 단계들에는 디폴트 구현이 있는 것이 도움이 될 수 있습니다. 자식 클래스들은 이러한 디폴트 메서드들을 구현할 필요가 없습니다.
4. 알고리즘의 중요한 단계들 사이에 흙들을 추가하는 것을 고려하세요.

5. 알고리즘의 각 변형에서 새로운 구상 자식 클래스를 생성하세요.
 새로운 구상 자식 클래스는 모든 추상 단계들을 반드시 구현해야 하지만 일부 선택 단계를 오버라이드할 수도 있습니다.

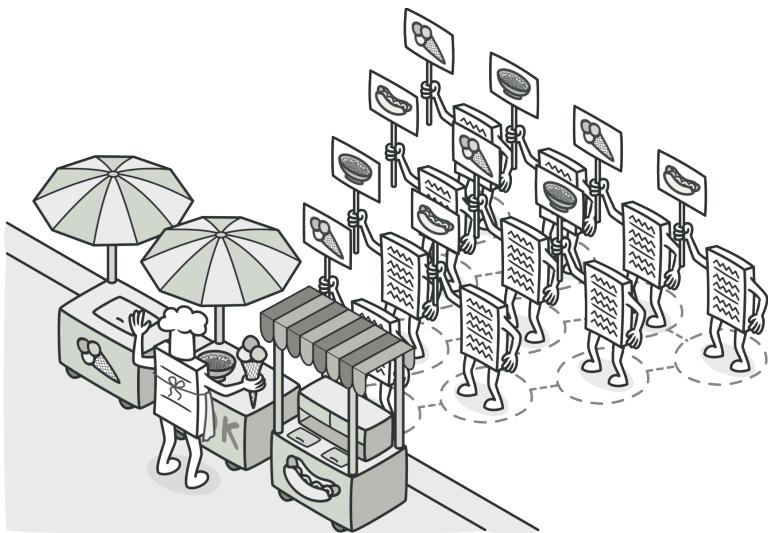
⚠ 장단점

- ✓ 클라이언트들이 대규모 알고리즘의 특정 부분만 오버라이드하도록 하여 그들이 알고리즘의 다른 부분에 발생하는 변경에 영향을 덜 받도록 할 수 있습니다.
- ✓ 중복 코드를 부모 클래스로 가져올 수 있습니다.
- ✗ 일부 클라이언트들은 알고리즘의 제공된 골격에 의해 제한될 수 있습니다.
- ✗ 당신은 자식 클래스를 통해 디폴트 단계 구현을 억제하여 리스트 코프 치환 원칙을 위반할 수 있습니다.
- ✗ 템플릿 메서드들은 단계들이 더 많을수록 유지가 더 어려운 경향이 있습니다.

↔ 다른 패턴과의 관계

- 팩토리 메서드는 템플릿 메서드의 특수화라고 생각할 수 있습니다. 동시에 대규모 템플릿 메서드의 한 단계의 역할을 팩토리 메서드가 할 수 있습니다.

- **템플릿 메서드는** 상속을 기반으로 합니다. 이 메서드는 자식 클래스들에서 알고리즘의 부분들을 확장하여 변경할 수 있도록 합니다. **전략 패턴은** 합성을 기반으로 합니다: 당신은 객체 행동의 일부분들을 이러한 행동에 해당하는 다양한 전략들을 제공하여 변경할 수 있습니다. **템플릿 메서드는** 클래스 수준에서 작동하므로 정적입니다. **전략 패턴은** 객체 수준에서 작동하므로 런타임에 행동들을 전환할 수 있도록 합니다.



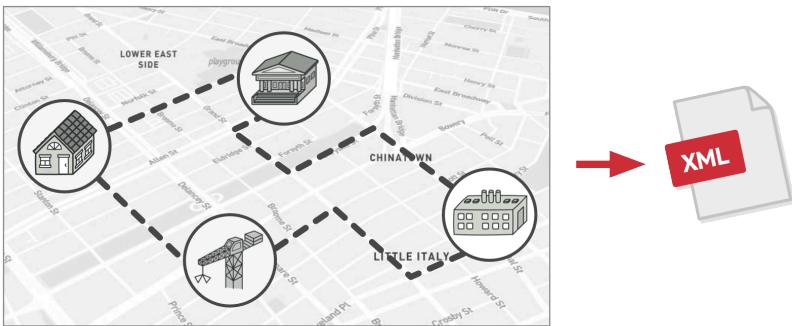
비지터 패턴

다음 이름으로도 불립니다: Visitor

비지터(방문자) 패턴은 알고리즘들을 그들이 작동하는 객체들로부터 분리할 수 있도록 하는 행동 디자인 패턴입니다.

(?): 문제

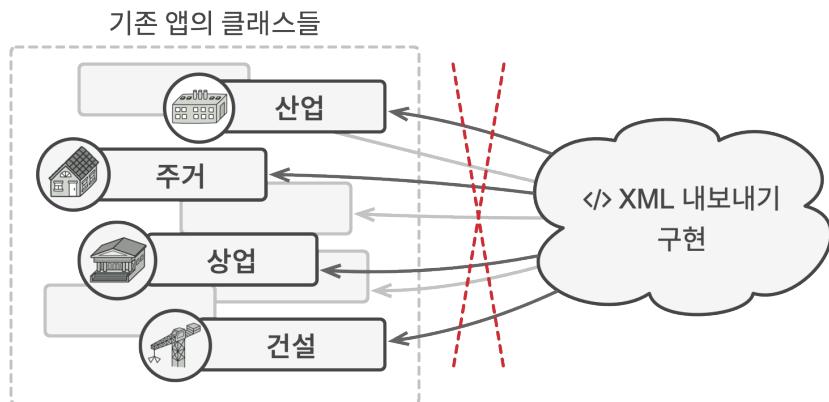
당신의 팀이 하나의 거대한 그래프로 구성된 지리 정보를 사용해 작동하는 앱을 개발하고 있다고 가정해 봅시다. 그래프의 각 노드는 도시와 같은 복잡한 객체를 나타낼 수 있지만 산업들, 관광 지역들 등의 더 세부적인 항목들도 나타낼 수 있습니다. 만약에 노드들이 나타내는 실제 객체들 사이에 도로가 있으면 노드들은 서로 연결됩니다. 각 노드 유형은 자체 클래스지만 각 노드는 객체입니다.



그래프를 XML 형식으로 내보내기

어느 날 당신은 그래프를 XML 형식으로 내보내는 작업을 구현하는 일을 맡았습니다. 처음에는 일이 매우 간단해 보였습니다. 당신은 각 노드 클래스에 내보내기 메서드를 추가한 다음 재귀를 활용하여 그래프의 각 노드에 작업하며 내보내기 메서드를 실행할 계획이었습니다. 해결책은 간단하고 우아했습니다. 다형성 덕분에 내보내기 메서드를 호출하는 코드를 노드들의 구상 클래스들에 결합하지 않았습니다.

불행히도 시스템의 설계자는 기존 노드 클래스들을 변경하는 것을 허용하지 않았습니다. 그는 코드가 이미 프로덕션 단계에 있으며 당신이 제안한 변경 사항들이 오류를 일으킬 수 있으므로 코드가 손상되는 위험을 감수하고 싶지 않다고 말했습니다.



XML 내보내기 메서드는 모든 노드 클래스에 추가되어야 했으며, 이러한 변경과 함께 버그가 발생하면 전체 앱이 망가질 위험이 있었습니다.

또 시스템 설계자는 노드 클래스들 내에 XML 내보내기 코드를 넣는 것이 적절한지에 대한 의문을 제기했습니다. 이 클래스들의 주 작업은 지리 데이터를 처리하는 것이므로, XML 내보내기 동작은 그곳에서 이상하게 보일 것이라고 했습니다.

시스템 설계자의 거절에는 또 다른 이유도 있었습니다. 위 기능이 구현된 후에도 마케팅 부서의 누군가가 데이터를 다른 형식으로 내보낼 수 있는 기능 또는 다른 기능을 요청할 가능성이 있으며,

그러면 당신은 다시 이 망가지기 쉬운 클래스들을 다시 한번 변경해야 한다는 것이었습니다.

😊 해결책

비지터 패턴은 당신이 새로운 행동을 기존 클래스들에 통합하는 대신 *visitor*(방문자)라는 별도의 클래스에 배치할 것을 제안합니다. 이제 행동을 수행해야 했던 원래 객체는 *visitor*의 메서드 중 하나에 인수로 전달됩니다. 그러면 메서드는 원래 객체 내에 포함된 모든 필요한 데이터에 접근할 수 있습니다.

이제 그 행동이 다른 클래스들의 객체들에 대해 실행될 수 있다면 어떨까요? 예를 들어 XML 내보내기의 경우 실제 구현은 다양한 노드 클래스들에서 약간씩 다를 수 있습니다. 따라서 비지터 클래스는 단일 메서드를 정의하는 대신 다음과 같이 메서드의 집합을 정의하여 각 메서드가 다른 유형의 인수를 받을 수 있도록 합니다:

```

1 class ExportVisitor implements Visitor {
2     method doForCity(City c) { ... }
3     method doForIndustry(Industry f) { ... }
4     method doForSightSeeing(SightSeeing ss) { ... }
5     // ...

```

그러나 우리는 이러한 메서드들을 정확히 어떻게 호출할까요? 특히 전체 그래프를 다룰 때 말입니다. 이 메서드들은

시그니처들이 다르므로 다양성을 사용할 수 없습니다. 주어진 객체를 처리할 수 있는 적절한 비지터 메서드를 선택하려면 먼저 그 클래스를 확인해야 합니다. 너무 복잡하지 않나요?

```

1  foreach (Node node in graph)
2      if (node instanceof City)
3          exportVisitor.doForCity((City) node)
4      if (node instanceof Industry)
5          exportVisitor.doForIndustry((Industry) node)
6      // ...
7  }
```

여기서 당신은 메서드 오버로딩을 사용하는 게 어떻겠냐고 제안할지도 모릅니다. 메서드 오버로딩은 다른 매개변수들의 집합들을 지원하더라도 모든 메서드에 같은 이름을 지정하는 방식입니다. 당신이 사용하는 프로그래밍 언어가 자바나 C#처럼 메서드 오버로딩을 지원한다고 가정하더라도, 그건 우리에겐 도움이 되지 않을 겁니다. 노드 객체의 정확한 클래스를 사전에 알 수 없으므로, 오버로딩 메커니즘은 실행해야 할 올바른 메서드가 무엇인지 판단할 수 없고, 따라서 디폴트(기본값)로 기초 Node 클래스의 객체를 받는 메서드를 선택하게 됩니다.

그러나 비지터 패턴에서는 이 문제를 더블 디스패치라는 방법을 사용하여 해결합니다. 이 방법은 번거로운 조건문 없이 객체에 적절한 메서드를 실행하는 것을 돋습니다. 클라이언트가 호출할 메서드의 적절한 버전을 선택하도록 하는 대신 이 선택권을

비지터에게 인수로 전달되는 객체에게 위임합니다. 이러한 객체들은 자신의 클래스들을 알고 있으므로 비지터에 대한 적합한 메서드를 더 쉽게 선택할 수 있습니다. 그들은 비지터를 '수락'하고 어떤 비지터 메서드가 실행되어야 하는지 알려줍니다.

```

1 // Client code
2 foreach (Node node in graph)
3     node.accept(exportVisitor)
4
5 // City
6 class City is
7     method accept(Visitor v) is
8         v.doForCity(this)
9         // ...
10
11 // Industry
12 class Industry is
13     method accept(Visitor v) is
14         v.doForIndustry(this)
15     // ...

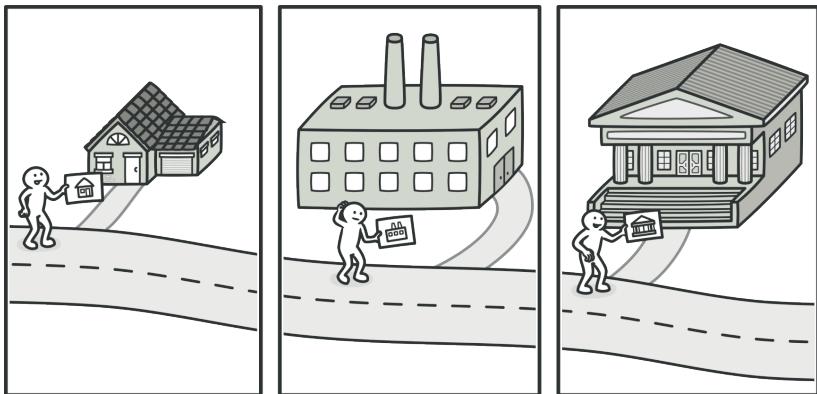
```

죄송합니다. 결국 노드 클래스들을 변경해야 했습니다. 그러나 최소한 변경 사항들은 사소했으며, 이제 코드를 다시 변경하지 않고도 다른 행동들을 추가할 수 있습니다.

이제 모든 비지터에 대한 공통 인터페이스를 추출하면 기존의 모든 노드가 당신이 앱에 도입하는 모든 비지터와 함께 작동할

수 있습니다. 노드와 관련된 새로운 행동을 도입하려면 새 비지터 클래스를 구현하기만 하면 됩니다.

☞ 실제상황 적용

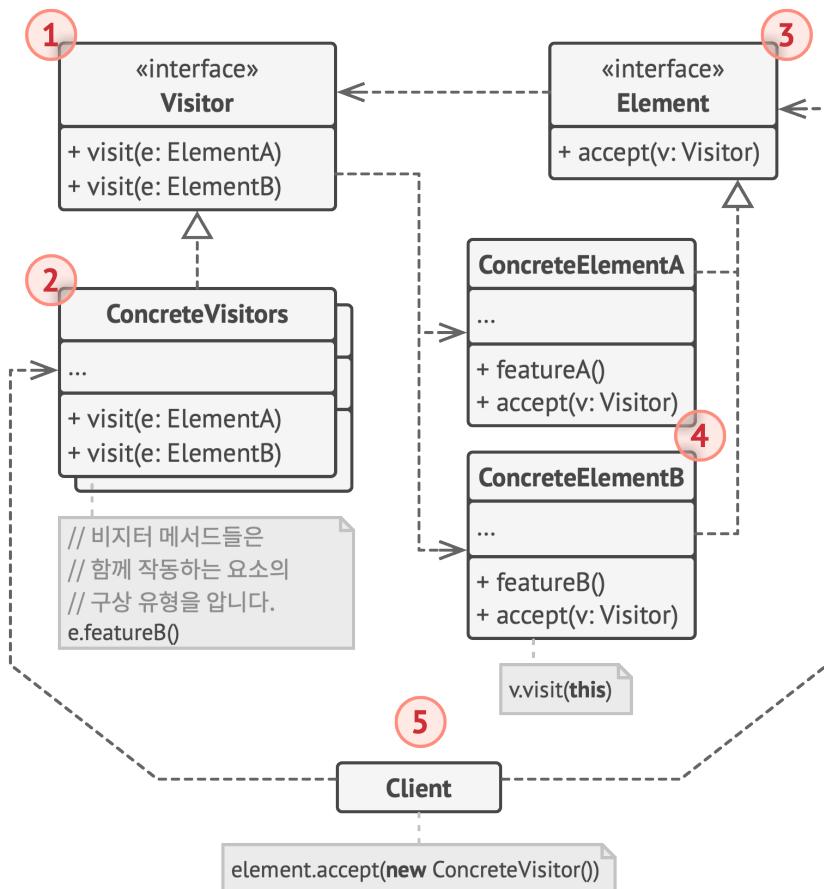


좋은 보험 대리인은 항상 다양한 유형의 조직들에 적절한 보험을 판매할 준비가 되어 있습니다.

새로운 고객을 확보하고 싶어 하는 노련한 보험 대리인을 상상해 봅시다. 그는 근방의 모든 건물을 방문하여 만나는 모든 사람에게 보험을 판매하려고 합니다. 그는 방문한 건물에 있는 회사 또는 조직의 유형에 따라 맞춤형 전문 보험 정책들을 제공할 수 있습니다.

- 주거용 건물을 방문할 때는 의료 보험을 판매합니다.
- 은행을 방문할 때는 도난 보험을 판매합니다.
- 커피숍을 방문할 때는 화재 및 홍수 보험을 판매합니다.

구조

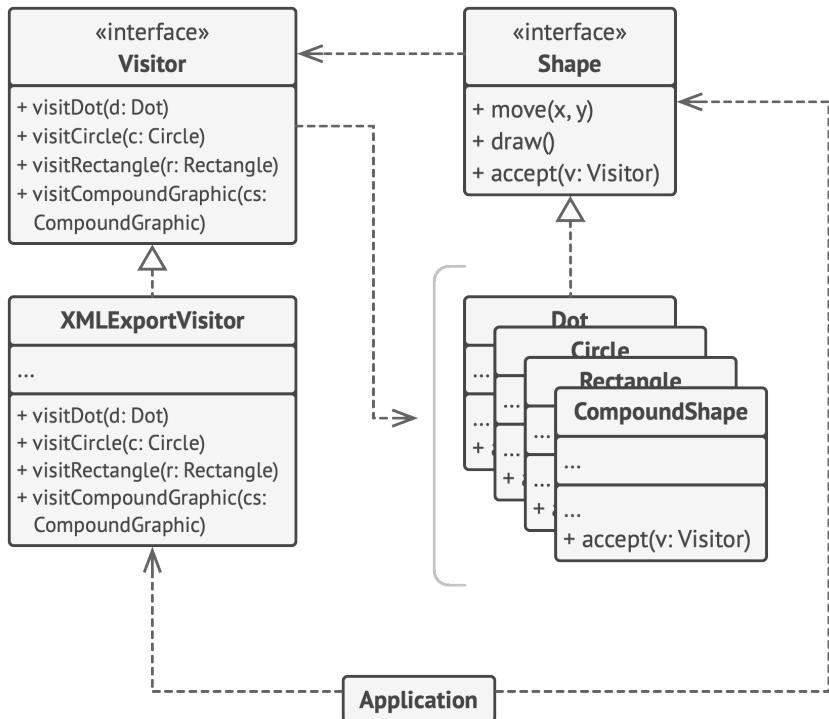


1. **비지터** 인터페이스는 객체 구조의 구상 요소들을 인수들로 사용할 수 있는 비지터 메서드들의 집합을 선언합니다. 이러한 메서드들은 (앱이 오버로딩을 지원하는 언어로 작성된 경우) 같은 이름을 가질 수 있지만 그들의 매개변수들의 유형은 달라야 합니다.

2. 각 **구상 비지터**는 다양한 구상 요소 클래스들에 맞춤으로 작성된 같은 행동들의 여러 버전을 구현합니다.
3. **요소** 인터페이스는 비지터를 '수락'하는 메서드를 선언합니다. 이 메서드에는 비지터 인터페이스 유형으로 선언된 하나의 매개변수가 있어야 합니다.
4. 각 **구상 요소**는 반드시 수락 메서드를 구현해야 합니다. 이 메서드의 목적은 호출을 현재 요소 클래스에 해당하는 적절한 비지터 메서드로 리다이렉트하는 것입니다. 기초 요소 클래스가 이 메서드를 구현하더라도 모든 자식 클래스들은 여전히 자신들의 클래스들 내에서 이 메서드를 오버라이드해야 하며 비지터 객체에 적절한 메서드를 호출해야 합니다.
5. **클라이언트**는 일반적으로 컬렉션 또는 기타 복잡한 객체(예: 복합체 트리)를 나타냅니다. 일반적으로 클라이언트들은 해당 컬렉션의 객체들과 어떠한 추상 인터페이스를 통해 작업하기 때문에 모든 구상 요소 클래스들을 인식하지 못합니다.

의사코드

이 예시에서의 **비지터** 패턴은 기하학적 모양들의 클래스 계층구조에 XML 내보내기 지원을 추가합니다.



비지터 객체를 통해 다양한 유형의 객체들을 XML 형식으로 내보내기.

```

1 // 요소 인터페이스는 기초 방문자 인터페이스를 인수로 받는 `accept` 메서드를
2 // 선언합니다.
3 interface Shape is
4     method move(x, y)
5     method draw()
6     method accept(v: Visitor)
7
8 // 각 구상 요소 클래스는 요소의 클래스에 해당하는 비지터의 메서드를 호출하는
9 // 방식으로 `accept` 메서드를 구현해야 합니다.
10 class Dot implements Shape is
11     // ...
12
  
```

```

13 // 참고로 우리는 현재 클래스 이름과 일치하는 `visitDot`를 호출하고
14 // 있습니다. 그래야 비지터가 함께 작업하는 요소의 클래스를 알 수 있습니다.
15 method accept(v: Visitor) is
16     v.visitDot(this)
17
18 class Circle implements Shape is
19     // ...
20     method accept(v: Visitor) is
21         v.visitCircle(this)
22
23 class Rectangle implements Shape is
24     // ...
25     method accept(v: Visitor) is
26         v.visitRectangle(this)
27
28 class CompoundShape implements Shape is
29     // ...
30     method accept(v: Visitor) is
31         v.visitCompoundShape(this)
32
33
34 // 비지터 인터페이스는 요소 클래스들에 해당하는 방문 메서드들의 집합을 선언합니다.
35 // 방문 메서드의 시그니처를 통해 비지터는 처리 중인 요소의 정확한 클래스를 식별할
36 // 수 있습니다.
37 interface Visitor is
38     method visitDot(d: Dot)
39     method visitCircle(c: Circle)
40     method visitRectangle(r: Rectangle)
41     method visitCompoundShape(cs: CompoundShape)
42
43 // 구상 비지터는 모든 구상 요소 클래스와 작동할 수 있는 같은 알고리즘의 여러 버전을
44 // 구현합니다.

```

```

45  //
46 // 비지터 패턴은 복합체 트리와 같은 복잡한 객체 구조와 함께 사용할 때 가장 큰
47 // 이득을 볼 수 있습니다. 그러면 비지터의 메서드들을 구조의 다양한 객체 위에서
48 // 실행하는 동안 알고리즘의 어떤 중간 상태를 저장하는 것이 도움이 될 수 있습니다.
49 class XMLExportVisitor implements Visitor is
50   method visitDot(d: Dot) is
51     // 점의 아이디와 중심 좌표를 내보냅니다.
52
53   method visitCircle(c: Circle) is
54     // 원의 아이디, 중심 좌표 및 반지름을 내보냅니다.
55
56   method visitRectangle(r: Rectangle) is
57     // 사각형의 아이디, 왼쪽 상단 좌표, 너비 및 높이를 내보냅니다.
58
59   method visitCompoundShape(cs: CompoundShape) is
60     // 모양의 아이디와 그 자식들의 아이디 리스트를 내보냅니다.
61
62
63 // 클라이언트 코드는 요소의 구상 클래스들을 파악하지 않고도 모든 요소 집합 위에서
64 // 비지터의 작업들을 실행할 수 있습니다. `accept` 작업은 비지터 객체의 적절한
65 // 작업으로 호출을 전달합니다.
66 class Application is
67   field allShapes: array of Shapes
68
69   method export() is
70     exportVisitor = new XMLExportVisitor()
71
72     foreach (shape in allShapes) do
73       shape.accept(exportVisitor)

```

이 예시에서 `accept` 메서드가 필요한 이유가 궁금하다면 제 설명글 [Visitor and Double Dispatch](#)에서 이 주제를 자세히 다루고 있습니다.

💡 적용

- ❖ **비지터 객체는 복잡한 객체 구조(예: 객체 트리)의 모든 요소에 대해 작업을 수행해야 할 때 사용하세요.**
- ❖ **비지터 패턴은 비지터 객체가 모든 대상 클래스들에 해당하는 같은 작업의 여러 변형들을 구현하도록 함으로써 다양한 클래스들을 가진 여러 객체의 집합에 작업을 실행할 수 있도록 해줍니다.**
- ❖ **비지터 패턴을 사용하여 보조 행동들의 비즈니스 로직을 정리하세요.**
- ❖ **이 패턴은 앱의 주 클래스들의 주 작업들을 제외한 모든 다른 행동들을 비지터 클래스들의 집합으로 추출함으로써 그들이 주 작업에 더 집중하도록 만들 수 있게 해줍니다.**
- ❖ **이 패턴은 행동이 클래스 계층구조의 일부 클래스들에서만 의미가 있고 다른 클래스들에서는 의미가 없을 때 사용하세요.**

- ⚡ 이 행동을 별도의 비지터 클래스로 추출한 후 관련 클래스들의 객체들을 수락하는 비지터 메서드들만 구현하고 나머지는 비워둡니다.

▣ 구현방법

1. 프로그램에 존재하는 각 구상 요소 클래스당 하나씩 '비지터 (방문)' 메서드를 만들고 이 메서드들의 집합으로 비지터 인터페이스를 선언하세요.
2. 요소 인터페이스를 선언하세요. 기존 요소 클래스 계층구조와 작업하는 경우 계층구조의 기초 클래스에 추상 수락 메서드를 추가하세요. 이 메서드는 비지터 객체를 인수로 받아들여야 합니다.
3. 모든 구상 요소 클래스들에서 수락 메서드들을 구현하세요. 이러한 메서드들은 단순히 비지터 메서드에 대한 호출을 들어오는 비지터 객체에 리다이렉트해야 합니다. 이 들어오는 비지터 객체는 현재 요소의 클래스와 일치합니다.
4. 요소 클래스들은 비지터 인터페이스를 통해서만 비지터와 작동해야 합니다. 그러나 비지터들은 비지터 메서드들의 매개변수 유형들로 참조된 모든 구상 요소 클래스들에 대해 알고 있어야 합니다.

5. 요소 계층구조 내에서 구현할 수 없는 각 행동의 경우, 새로운 구상 비지터 클래스를 만들고 모든 비지터 메서드들을 구현하세요.

비지터가 요소 클래스의 일부 비공개 필드들 또는 메서드들에 접근해야 할 상황이 발생할 수 있습니다. 이럴 때 이러한 필드들 또는 메서드들을 공개하여 요소의 캡슐화를 위반하거나, 비지터 클래스를 요소 클래스에 중첩할 수 있습니다. 중첩 옵션의 경우 중첩 클래스들을 지원하는 프로그래밍 언어를 사용할 때만 가능합니다.

6. 클라이언트는 비지터 객체들을 만들고 '수락' 메서드들을 통해 그들을 요소들에 전달해야 합니다.

△△ 장단점

- ✓ **개방/폐쇄 원칙.** 당신은 다른 클래스를 변경하지 않으면서 해당 클래스의 객체와 작동할 수 있는 새로운 행동을 도입할 수 있습니다.
- ✓ **단일 책임 원칙.** 같은 행동의 여러 버전을 같은 클래스로 이동할 수 있습니다.
- ✓ 비지터 객체는 다양한 객체들과 작업하면서 유용한 정보를 축적할 수 있습니다. 이것은 객체 트리와 같은 복잡한 객체 구조를 순회하여 이 구조의 각 객체에 비지터 패턴을 적용하려는 경우에 유용할 수 있습니다.

- ✖ 당신은 클래스가 요소 계층구조에 추가되거나 제거될 때마다 모든 비지터를 업데이트해야 합니다.
- ✖ 비지터들은 함께 작업해야 하는 요소들의 비공개 필드들 및 메서드들에 접근하기 위해 필요한 권한이 부족할 수 있습니다.

↔ 다른 패턴과의 관계

- **비지터** 패턴은 **커맨드** 패턴의 강력한 버전으로 취급할 수 있습니다. 비지터 패턴의 객체들은 다른 클래스들의 다양한 객체에 대한 작업을 실행할 수 있습니다.
- 당신은 **비지터** 패턴을 사용하여 **복합체** 패턴 트리 전체를 대상으로 작업을 수행할 수 있습니다.
- **비지터** 패턴과 **반복자** 패턴을 함께 사용해 복잡한 데이터 구조를 순회하여 해당 구조의 요소들의 클래스들이 모두 다르더라도 이러한 요소들에 대해 어떤 작업을 실행할 수 있습니다.

결론

축하합니다! 책을 끝마쳤습니다!

그러나 세상에는 다른 패턴들도 많습니다. 이 책이 당신이 패턴을 학습하고 영웅 수준의 프로그램 디자인 능력을 개발하는 데 있어 출발점이 되기를 바랍니다. 여기서는 당신이 다음에 무엇을 공부할지 결정할 수 있도록 몇 가지를 제안해 보았습니다.

- </> 여러 프로그래밍 언어들로 된 코드 예시들이 있는 저장소를 사용하실 수 있다는 점을 잊지 마세요. 당신의 계정에서 예시들을 다운받을 수 있습니다.

<https://refactoring.guru/home>

- ☒ 조슈아 케리에브스키의 책 패턴을 활용한 리팩토링을 읽어보세요.

<https://refactoring.guru/ko/ref-to-patterns-book>

- 🔍 리팩토링에 대해 아무것도 모르시나요? 제 리팩토링에 뛰어들기 교육과정을 살펴보세요.

<https://refactoring.guru/ko/refactoring/course>

- ☐ 이 패턴 치트 시트를 인쇄하여 항상 볼 수 있는 곳에 두세요.

<https://refactoring.guru/ko/design-patterns/cheatsheets>

- 💬 마지막으로, 저는 (매우 비판적일지라도 😊) 이 책에 대한 당신의 피드백을 매우 감사하게 생각합니다. 피드백은 여기에 남겨주세요:

<https://refactoring.guru/ko/refactoring/feedback>