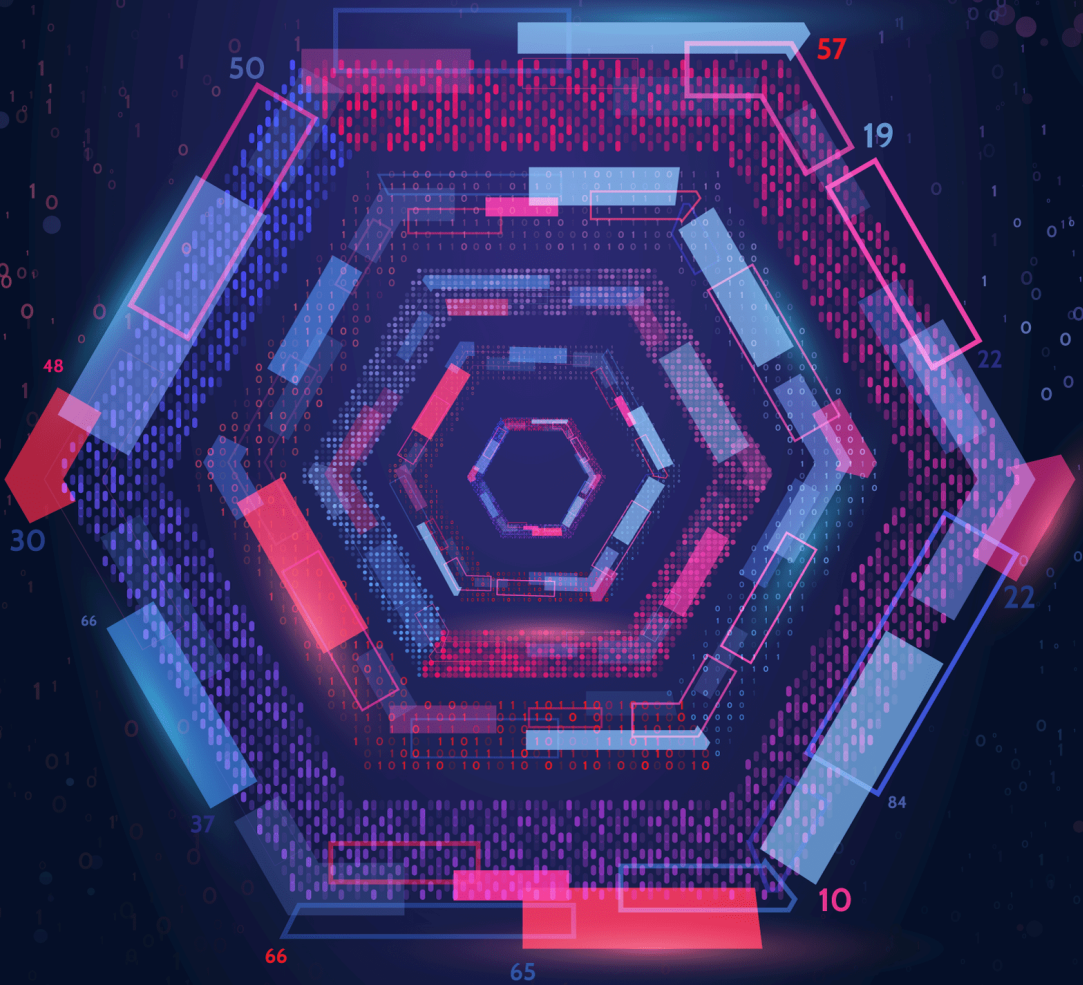
디자인 패턴에 뛰어들기



알렉산더 슈베츠

디자인 패턴에 뛰어들기

v2022-1.1

데모 버전

저작권에 대한 몇 마디

안녕하세요! 제 이름은 알렉산더 슈베츠입니다. 저는 『디자인 패턴에 뛰어들기[1]』 라는 책과 『리팩토링에 뛰어들기[2]』 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.

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

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 앵커들을 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 용어들**을 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.

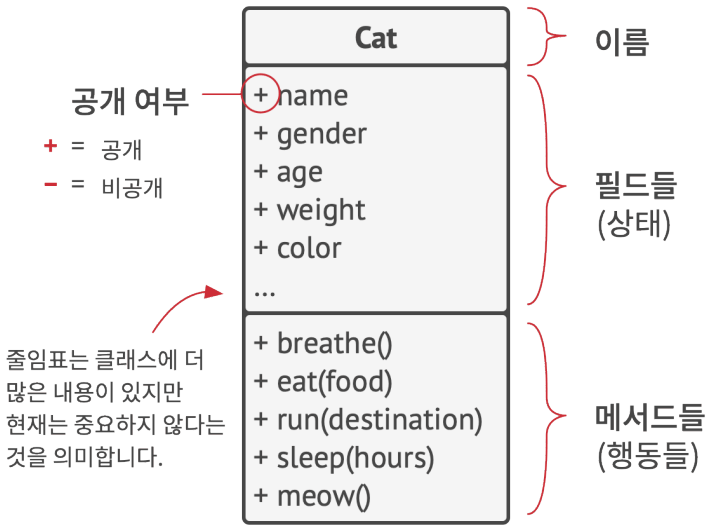
객체 지향
프로그래밍 소개

OOP의 기초

간략히 OOP라고도 불리는 **객체 지향 프로그래밍**(***O**bject **O**riented **P**rograming*)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.

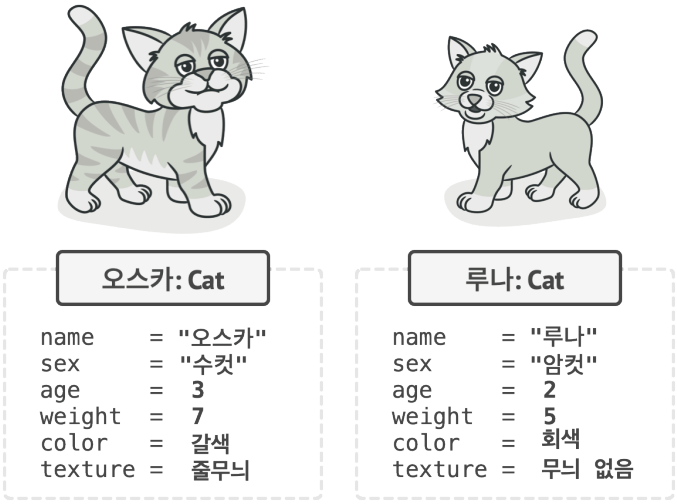
> 이 책에서 저는 클래스 이름을 UML 다이어그램이나 코드에 쓰인 것처럼 영어로 언급할 겁니다. 하지만 때로는 글이 대화처럼 읽힐 수 있게 클래스 이름을 한국어로 번역하여 언급할 수도 있으니 유념해 주세요.

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.



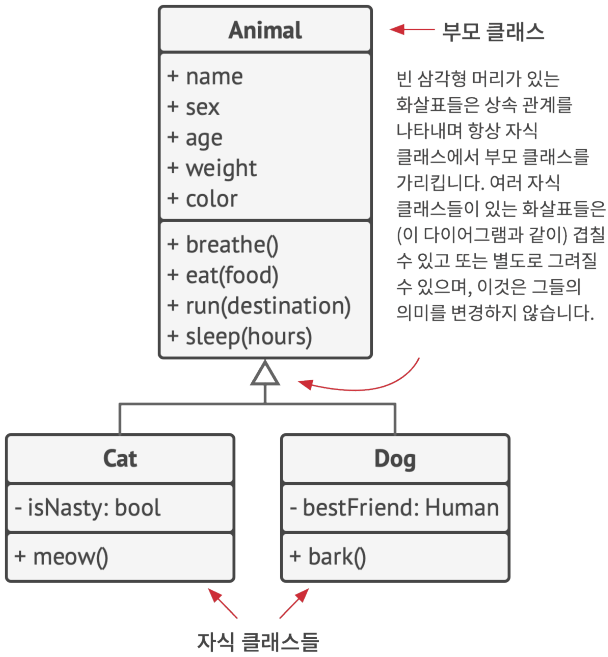
객체들은 클래스들의 인스턴스들입니다.

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.

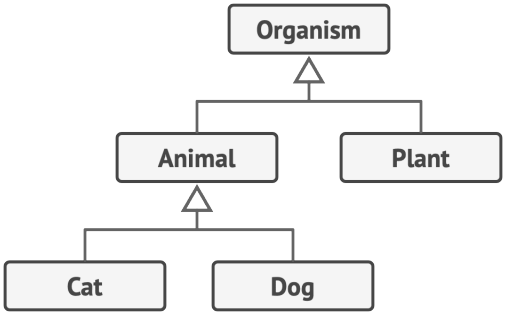


클래스 계층구조의 UML 다이어그램. 이 다이어그램의 모든 클래스는 `Animal` (동물) 클래스 계층구조의 일부입니다.

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.

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 감마[1]의 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.

```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  if (order.country == "US")
    total += total * 0.07 // US sales tax
  else if (order.country == "EU"):
    total += total * 0.20 // European VAT

  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.

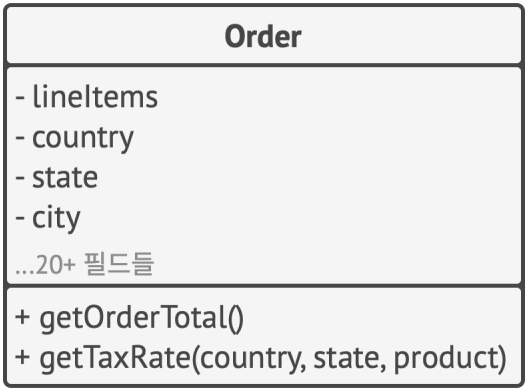
```
method getOrderTotal(order) is
  total = 0
  foreach item in order.lineItems
    total += item.price * item.quantity

  total += total * getTaxRate(order.country)

  return total

method getTaxRate(country) is
  if (country == "US")
    return 0.07 // US sales tax
  else if (country == "EU")
    return 0.20 // European VAT
  else
    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.

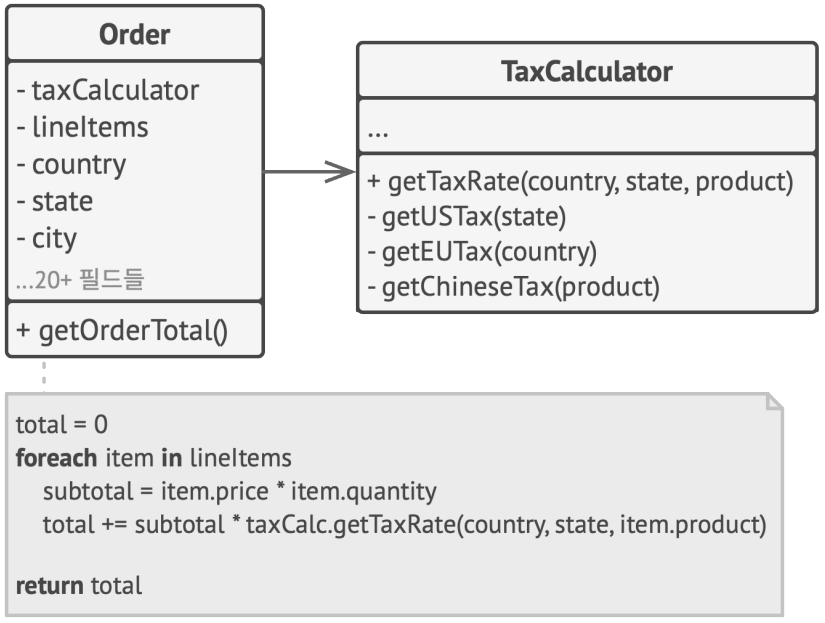
클래스 수준에서의 캡슐화

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.



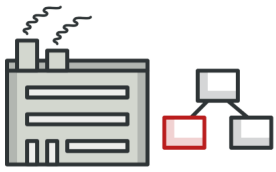
수정 후: 세금 계산은 `Order` (주문) 클래스로부터 숨겨집니다.

책의 유료 정식 버전의 30페이지가
데모 버전에서 생략됩니다.

디자인 패턴 목록

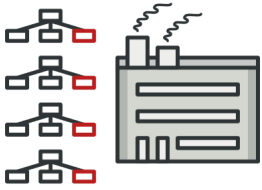
생성 디자인 패턴

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.

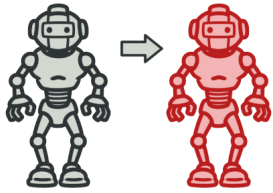


추상 팩토리

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

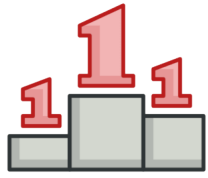
빌더

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



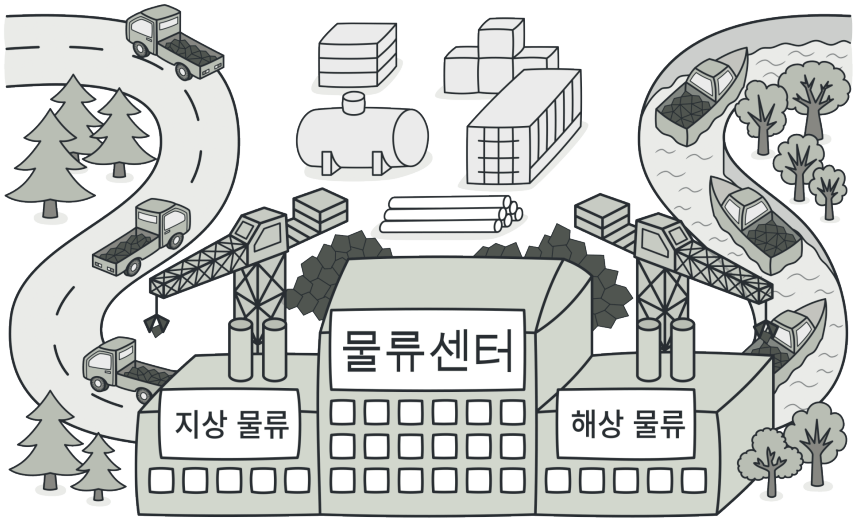
프로토타입

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.

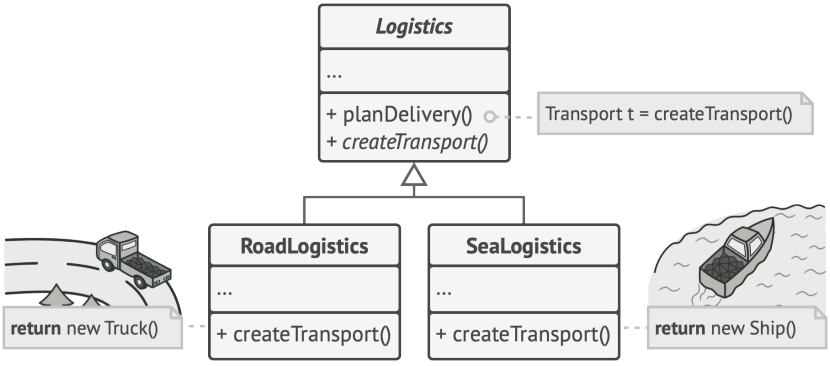
나머지 코드가 이미 기존 클래스들에 결합되어 있다면 프로그램에 새 클래스를 추가하는 일은 그리 간단하지 않습니다.

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 제품이라고도 불립니다.

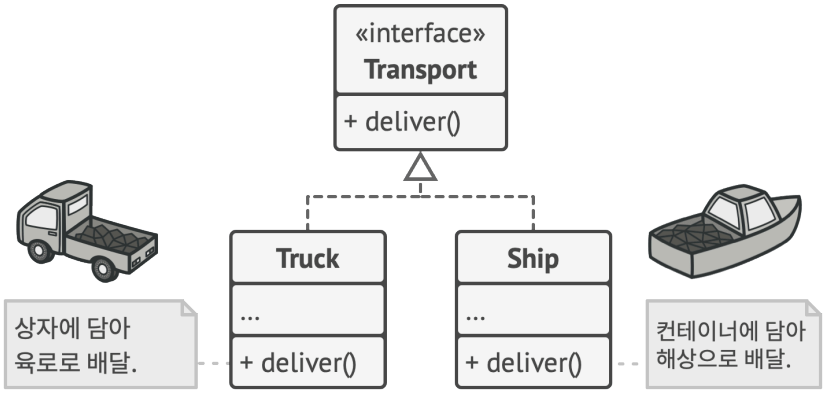


자식 클래스들은 팩토리 메서드가 반환하는 객체들의 클래스를 변경할 수 있습니다.

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.

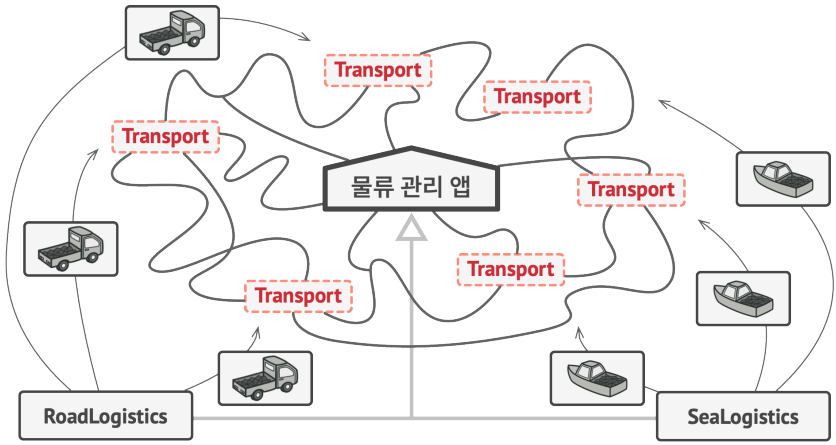
하지만 약간의 제한이 있긴 합니다. 자식 클래스들은 다른 유형의 제품들을 해당 제품들이 공통 기초 클래스 또는 공통 인터페이스가 있는 경우에만 반환할 수 있습니다. 또 이전에 언급한 모든 제품들에 공통인 `Transport` 인터페이스로 `Logistics` 기초 클래스의 `createTransport` 팩토리 메서드의 반환 유형을 선언해야 합니다.



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)

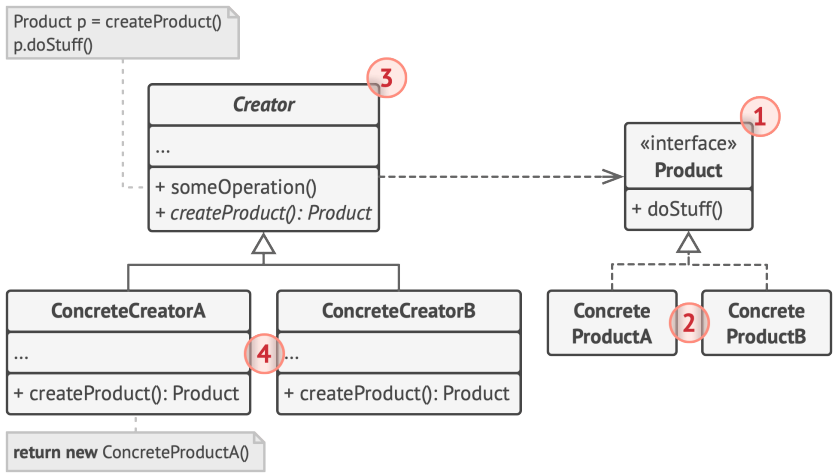
클래스에 포함된 팩토리 메서드는 `Truck` 객체들을 반환하는 반면 `SeaLogistics` (해운 물류) 클래스에 포함된 팩토리 메서드는 선박 객체들을 반환합니다.



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



1. **제품**은 인터페이스를 선언합니다. 인터페이스는 생성자와 자식 클래스들이 생성할 수 있는 모든 객체에 공통입니다.

2. **구상 제품들**은 제품 인터페이스의 다양한 구현들입니다.

3. **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 디폴트(기본값) 제품 유형을 반환하도록 만들 수도 있습니다.

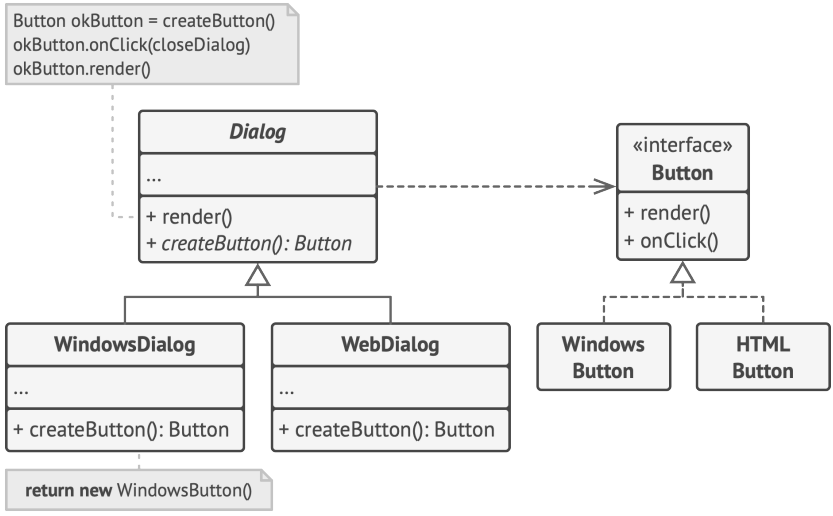
크리에이터라는 이름에도 불구하고 크리에이터의 주책임은 제품을 생성하는 것이 **아닙니다.** 일반적으로 크리에이터 클래스에는 이미 제품과 관련된 핵심 비즈니스 로직이 있으며, 팩토리 메서드는 이 로직을 구상 제품 클래스들로부터 디커플링(분리) 하는 데 도움을 줄 뿐입니다. 대규모 소프트웨어 개발사에 비유해 보자면, 이 회사는 프로그래머들을 위한 교육 부서가 있을 수 있으나, 회사의 주 임무는 프로그래머를 교육하는 것이 아니라 코드를 작성하는 것입니다.

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

   참고로 팩토리 메서드는 항상 새로운 인스턴스들을 **생성**해야 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 다이얼로그(대화 상자) 예시.

기초 `Dialog`(대화 상자) 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 대화 상자 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, 대화상자에 새로운 팩토리 메서드를 추가할 때마다 이 프로그램은 **추상 팩토리** 패턴에 더 가까워집니다. 걱정하지 마세요. 추상 팩토리 패턴에 대해서는 나중에 설명하겠습니다.

```
// 크리에이터 클래스는 제품 클래스의 객체를 반환해야 하는 팩토리 메서드를
// 선언합니다. 크리에이터의 자식 클래스들은 일반적으로 이 메서드의 구현을
// 제공합니다.
class Dialog is
  // 크리에이터는 팩토리 메서드의 일부 디폴트 구현을 제공할 수도 있습니다.
  abstract method createButton():Button

  // 크리에이터의 주 업무는 제품을 생성하는 것이 아닙니다. 크리에이터는
  // 일반적으로 팩토리 메서드에서 반환된 제품 객체에 의존하는 어떤 핵심
  // 비즈니스 로직을 포함합니다. 자식 클래스들은 팩토리 메서드를 오버라이드 한
  // 후 해당 메서드에서 다른 유형의 제품을 반환하여 해당 비즈니스 로직을
  // 간접적으로 변경할 수 있습니다.
  method render() is
    // 팩토리 메서드를 호출하여 제품 객체를 생성하세요.
    Button okButton = createButton()
    // 이제 제품을 사용하세요.
    okButton.onClick(closeDialog)
    okButton.render()

```

```
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
```

```
  field dialog: Dialog

  // 앱은 현재 설정 또는 환경 설정에 따라 크리에이터의 유형을 선택합니다.
  method initialize() is
    config = readApplicationConfigFile()

    if (config.OS == "Windows") then
      dialog = new WindowsDialog()
    else if (config.OS == "Web") then
      dialog = new WebDialog()
    else
      throw new Exception("Error! Unknown operating system.")

  // 클라이언트 코드는 비록 구상 크리에이터의 기초 인터페이스를 통하는 것이긴
  // 하지만 구상 크리에이터의 인스턴스와 함께 작동합니다. 클라이언트가
  // 크리에이터의 기초 인터페이스를 통해 크리에이터와 계속 작업하는 한 모든
  // 크리에이터의 자식 클래스를 클라이언트에 전달할 수 있습니다.
  method main() is
    this.initialize()
    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 둥근 버튼들이 필요한데 프레임워크는 사각형 버튼만 제공한다고 가정합시다. 또 표준 `Button`(버튼) 클래스는 `RoundButton`(둥근 버튼) 자식 클래스로 확장했지만, 이제 메인 `UIFramework`(사용자 인터페이스 프레임워크) 클래스에 디폴트 클래스 대신 새로운 `RoundButton`(둥근 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 메서드**로 시작해 더 유연하면서도 더 복잡한 **추상 팩토리**, **프로토타입** 또는 **빌더** 패턴으로 발전해 나갑니다.

- **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.

- **팩토리 메서드**를 **반복자**와 함께 사용하여 컬렉션 자식 클래스들이 해당 컬렉션들과 호환되는 다양한 유형의 반복자들을 반환하도록 할 수 있습니다.

- **프로토타입**은 상속을 기반으로 하지 않으므로 상속과 관련된 단점들이 없습니다. 반면에 프로토타입은 복제된 객체의 복잡한 초기화가 필요합니다. **팩토리 메서드**는 상속을 기반으로 하지만 초기화 단계가 필요하지 않습니다.

- **팩토리 메서드**는 **템플릿 메서드**의 특수화라고 생각할 수 있습니다. 동시에 대규모 템플릿 메서드의 한 단계의 역할을 팩토리 메서드가 할 수 있습니다.

책의 유료 정식 버전의 343페이지가
데모 버전에서 생략됩니다.